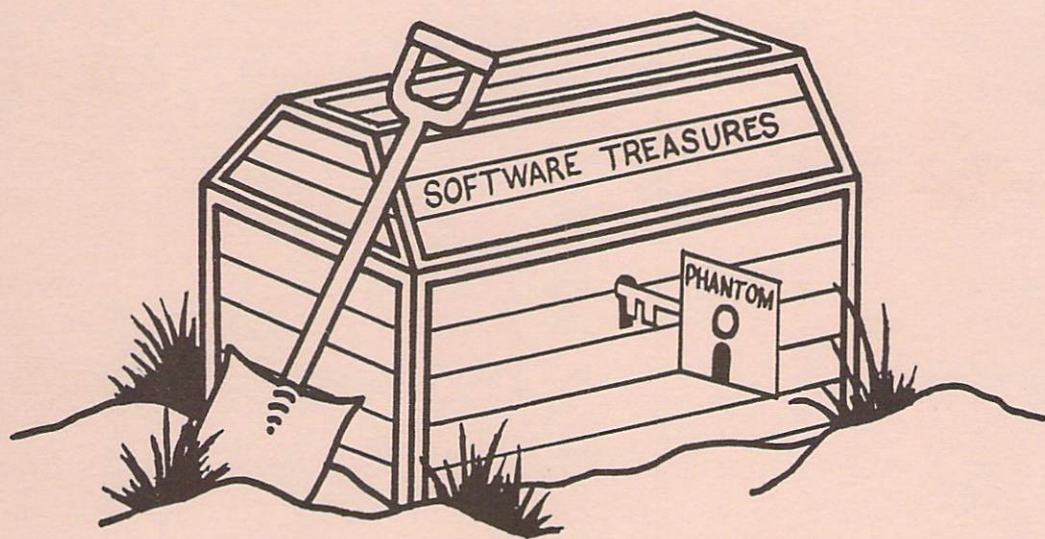
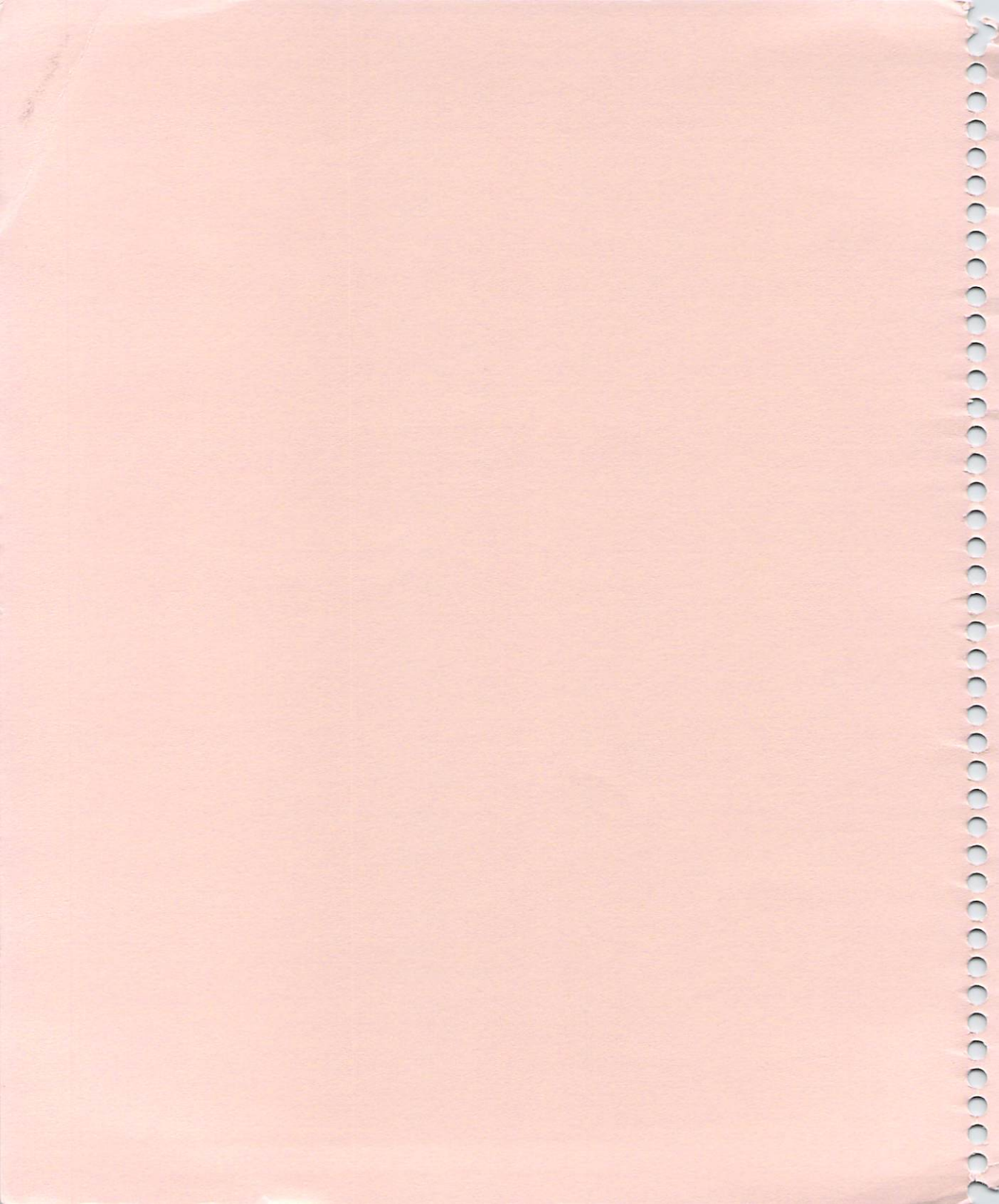


CSM NEWSLETTER

Compendium



(c) Copyright 1939
Hands on Software Inc.
All Rights Reserved



INTRODUCTION

We at Kracker Jax are proud to present the best features of one of the finest newsletters ever written on the subject of Commodore Computers. This Compendium represents countless hours of investigation and research. In the years that the CSM Newsletter was produced, it was widely recognized as **THE** source of insider information, and most subscribers waited impatiently each month for their issue to appear in their mailbox.

In these pages, you will find information on hardware projects, how nybbblers work, custom ROMs for your computer, inside programming techniques, 1541 drive programming, just for fun - key in programs, and many, many protection schemes unraveled before your very eyes. The hours you'll spend with this manual will be some of the most educational, yet fun. Although CSM's heyday was in the years 1984 through 1986, you'll find that most of the information presented, still holds water today.

As you read this manual, you'll see references to nybbblers, monitors, sector editors and other software based programs. For those of you who own the Maverick utility package, you should find all the tools needed within. For those who haven't yet purchased the Maverick, we highly recommend it.

For those of you who own the RAMBoard, please note that the 8K Software detailed within should work fine. Also, the 8K Disk Drive RAM project in this compendium will provide you with a suitable RAMBoard substitute while using the Maverick.

Also, you'll see countless references to the CSM Protection Manuals 1 and 2. Rather than see the source of these books completely "dry up", we have purchased the rights to these fine reference manuals and highly recommend them as further reading. These manuals are available from Software Support at the following prices:

CSM Protection Manual One : \$24.95

CSM Protection Manual Two : \$29.95

Also Available :

Kracker Jax Revealed Vol One, Two, or Three : \$23.50 each

The Maverick : \$34.95

The RAMBoard (8K Drive RAM) : \$34.95

U.S. customers, please add \$3.50 for S/H per order. Overseas customers please write for exact charges.

DISTRIBUTED BY
SOFTWARE SUPPORT
2700 N.E. ANDRESEN RD. # A-1
VANCOUVER WA 98661
(206) 695-9648

NEWSLETTER INDEX, 1984-86

TECHNICAL / UTILITIES	ISSUE	PG. No.
1541 EXTRA 8K/16K RAM	Bonus	5
8K TRACK READ PROGRAM	BONUS	14
1541 HARDWARE LIMITS	April 85	15
1541 BIT COPYING, Part 1	Jan. 86	19
1541 BIT COPYING, Part 2	Feb. 86	21
1541 FREEZE (FOR 8K RAM)	July 86	23
1541-ONLY RESET SWITCH	Nov. 85	27
1571 REVISIONS	May. 86	28
ALTERNATE KERNAL CALLS	Aug. 85	29
AMPERSAND (&) FILES	Aug. 85	32
C-128 COMPUTER	Jan. 86	35
CUSTOM KERNAL	June 85	37
NIBBLING UNPROT. DISKS	Oct. 85	39
EXECUTING 1541 ROUTINES	June 85	40
FAST LOADERS	May 86	43
KERNAL ROM REVISIONS	May 85	45
MODIFYING DOS (General)	Oct. 84	47
MODIFYING DOS (40 Trks)	Dec. 84	48
SAVE @ BUG	Nov. 85	50
ULTRA LIGHTNING LOADER	April 85	50
UNIVERSAL BACKUP PROGRAM	April 86	52
USING SNAPSHOT 64	Nov. 85	52
WORKING INSIDE 1541	Feb. 85	54
ADJACENT HALF-TRACKS	June 85	58
WR PROT / DEV # SWITCHES	Oct. 85	59

PROGRAM BACKUP PROCEDURES	PG No.	COMPANY	ISSUE
ACROJET	62	MicroProse Software	May 86
ADVENTURE MASTER	63	CBS Software	March 85
BANK STREET MUSICWRITER	64	Mindscape, Inc.	May 85
BANK STREET WRITER	64	Bank St. College	April 86
BC'S QUEST FOR TIRES	66	Sierra On-Line Inc.	Aug. 84
BEACH-HEAD II	67	Access Software	Nov. 85
BELOW THE ROOT	68	Windham Classics	July 86
BLITZ! COMPILER	69	Prolic / Skyles	July 85
C POWER V2.6	70	Pro-Line Software	Dec. 85
CHAMPIONSHIP BOXING	72	Sierra On-Line	Feb. 86
COHEN'S TOWER	73	Datamost / Fanda	Feb. 85
COPY-Q II	74	Q-R&D	July 85
COUNTDOWN TO SHUTDOWN	75	Activision	Jan. 86
CREATIVE FILER	76	Creative Software	July 85
CROSSWORD MAGIC	79	Mindscape, Inc.	Jan. 86
CRYPTO CUBE	80	DesignWare, Inc.	April 86
DECATHALON	81	Activision	Nov. 84
DEVELOP-64 4.0	83	Don French	Aug. 85

DONALD DUCK'S PLAYGROUND	85	Sierra On-Line Inc.	Sept. 85
EASY SCRIPT	86	Precision Software	Jan. 85
ECONOMICS	89	Micro-Ed, Inc.	April 85
F-15 STRIKE EAGLE	90	MicroProse Software	May 85
F-15 STRIKE EAGLE Update	92	MicroProse Software	June 85
FACTORY, THE	92	Sunburst Comm.	April 85
FLEET SYSTEM 3 (C128)	93	Professional S'ware	June 86
FLIGHT SIMULATOR II	95	SubLOGIC Corp.	Oct. 85
GARFIELD	97	Develop. Learning	July 86
GEOS V1.0	98	Berkeley Softworks	July 86
GHOST CHASER	101	Fanda	Oct. 85
HEART OF AFRICA (SNAPSHOT)	103	Electronic Arts	Dec. 85
HEIST, THE	104	Mike Livesay Games	Sept. 84
HESMON 64 (Cartridge)	105	H.E.S.	Oct. 84
HESMON 64 Modification	107	H.E.S.	Nov. 84
IMPERIUM GALACTUM	107	SSI	Nov. 85
JUPITER MISSION 1999	108	Avalon Hill Game Co.	March 86
KIDS ON KEYS	111	Spinnaker Corp.	April 85
KOALAPainter	112	Koala Technologies	June 85
LEADER BOARD GOLF, Part 1	113	Access Software	May 86
LEADER BOARD GOLF, Part 2	115	Access Software	June 86
LODERUNNER	118	Broderbund	Sept. 84
LODERUNNER Addendum	120	Broderbund	Oct. 84
MAGIC DESK (Cartridge)	121	Commodore	Dec. 84
MASTER OF THE LAMPS	126	Activision	Sept. 85
MBA-TOR (27.DEC.84)	128	AEA, Inc.	July 85
MICRO COOKBOOK	131	Commodore	Sept. 85
MICROSOFT MULTIPLAN	132	HesWare / Microsoft	Oct. 84
MILLIONAIRE V2.0	134	Blue Chip Software	Feb. 86
MONTEZUMA'S REVENGE	136	Parker Brothers	June 85
MORSE UNIVERSITY 6/11/85	137	AEA, Inc.	March 86
MOVIE MAKER	139	Interactive Pictures	June 85
MSD CLONE MACHINE	140	Micro-Ware Distrib.	June 86
MUSIC PROCESSOR	141	Sight & Sound Music	May 85
MUSIC SHOP, THE	144	Broderbund	Feb. 86
NEWSROOM	145	Springboard Software	May 86
NEWSROOM Update	147	Springboard Software	Jan. 86
NIGHT MISSION PINBALL	147	SubLOGIC Corp.	Nov. 84
NY TIMES CROSSWORD VOL. 1	149	Softie, Inc.	Nov. 85
OMNICALC	150	ISA Systems, Inc.	Oct. 84
ON-COURT TENNIS	151	Gamestar	Feb. 85
OXFORD PASCAL	152	OCSS & D. Goodman	June 85
PANZER BRIGADIER	155	SSI	June 86
PFS FILE	155	Software Publ. Corp.	Aug. 85
PHAROAH'S CURSE	156	Synapse Software	Sept. 84
PIECE OF CAKE	157	Counterpoint S'ware	April 85
POKER SAM	158	Don't Ask Software	Oct. 85
PROFESSIONAL TOUR GOLF	160	SSI	May 85
PROMAL	161	Softspoken, Inc.	Aug. 85
QUEST, THE	161	Penguin Software	Nov. 84

QUESTRON	163	Charles Dougherty	March 86
REVERSAL	164	Hayden Book Company	Aug. 84
ROCKY'S BOOTS	165	Learning Company	March 85
SAFEGUARD 64 DEMO	166	Glenco Engineering	May 85
SARGON III	168	Hayden Software	Oct. 84
SCROLLS OF ABADON	169	Access Software	Jan. 85
SIDEWAYS	172	Timeworks / Funk	Nov. 85
SKY TRAVEL/BGRAPH	173	Commodore / Deltron	Dec. 85
SNOKIE	174	Funsoft, Inc.	March 86
SPACE TAXI	176	Kutcher / Muse	Aug. 85
SPELLICOPTER	179	DesignWare, Inc.	April 85
SPELLMASTER	180	Systems Software	Jan. 85
SPY VS. SPY	181	First Star Software	March 85
SUICIDE STRIKE	182	Tronix	Dec. 84
TRIO	184	Softsync, Inc.	May 86
TRIVIA	185	Cymbal Software Inc.	Aug. 84
ULTIMA III	186	Origin Systems	Dec. 84
ULTRACOPY-II	187	Ultrabyte	Dec. 85
UNGUARD (CLONE MACHINE)	188	Micro-Ware Distrib.	Sept. 84
UP'N DOWN	190	Sega	Feb. 85
VIP TERM	191	Softlaw Corp.	June 85
VIP TERM Addendum	192	Softlaw Corp.	Sept. 85
VISIBLE SOLAR SYSTEM(Cart)	192	Commodore	March 85
WIZARD	194	Prog. Peripherals	Nov. 84
WIZARD Addendum	197	Prog. Peripherals	Dec. 84
WIZARD Addendum 2	198	Prog. Peripherals	April 85
WORDPRO 3 PLUS	198	Professional S'ware	Jan. 85
WORDPRO 128	199	Pro-Line Software	April 86
YOUR NET WORTH	201	ISA Systems, Inc.	May 85

TECHNICAL/UTILITIES

EXTRA 8K RAM FOR THE 1541

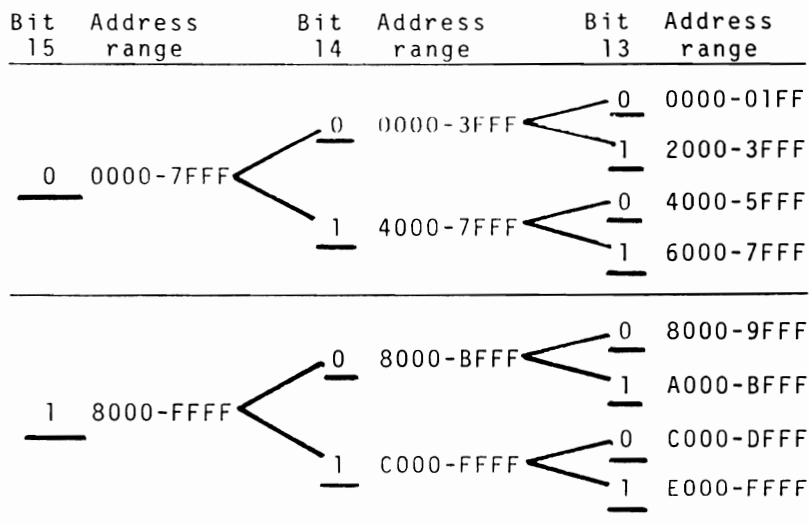
THEORY OF OPERATION

The standard 1541 disk drive contains 16K of ROM memory and 2K of RAM memory. The 16K of ROM memory is divided between two 8K ROM chips, located at \$C000-DFFF and \$E000-FFFF in the drive's memory map. The 2K of RAM is located at \$0000-\$07FF and is usually resident on one 2016-type chip. On older, 'long-board' drives, the 2K is supplied by four 2114-type chips.

Each memory chip has a +5 volt (power) connection and a ground connection. Each chip also has address and data line connections for use when the chip is being accessed by the microprocessor. The address lines are used to tell the chip which of its bytes is being accessed, and the data lines are used to send or receive the actual data byte. The RAM chip has another line called the READ/WRITE (R/W) OR WRITE ENABLE (WE) line. When this line is high, it indicates a read operation; when it is low it indicates a write operation. The ROM chips don't have a R/W line since they are READ ONLY.

Both types of memory chips also have a control line called the CHIP ENABLE (CE) OR CHIP SELECT (CS) line. This line is like an 'on/off' switch for the chip; if it is held high (+5), the chip will effectively be turned off. Only when this line is held low (grounded) will the chip respond to access requests through the address and data lines. 'Address decoding' circuitry in the drive uses this CS line to 'turn on' different chips, depending on which address the microprocessor is trying to access. The address decoder selects the proper memory chip based on the highest bits of the address, and the chip itself selects the proper byte based on the lower bits. For example, whenever an address in the range \$0000-07FF is accessed, the address decoder will select the 2K RAM chip, and the RAM chip will decode the rest of the address internally. This is how different chips are located at different addresses in the drive's memory map. See the chapter 'The 6510 and the PLA' in the P.P.M. Vol. II for a discussion of memory management.

We will add our 8K of extra RAM memory by 'patching' into the address, data, R/W and CS lines. We will also need to do some of our own address decoding. Our extra RAM will reside at \$A000-BFFF in the drive's memory map. Any time the microprocessor references an address in this range it is a sign to select our RAM chip. Our address decoder will detect a memory reference in this range by examining the three highest bits of the address (bits 15, 14 and 13). The highest bit (bit 15) specifies whether the address is in the bottom half of memory (\$0000-7FFF) or the top half (\$8000-FFFF). All addresses under \$8000 have a 0 in bit 15, and all addresses equal to or greater than \$8000 have a 1 in bit 15. Likewise, the next highest bit (bit 14) divides each of these ranges into two halves, and so on. The following chart shows the ranges specified by each combination of bits 13-15.



From the above chart, you can see that addresses in the range of \$A000-BFFF always start with bits 15,14,13 = 101. Note also that addresses in the range \$E000-FFFF start with 111. Thus bit 14 can be used to distinguish between these two ranges of addresses. Unfortunately, the address decoding already in the drive does not use bit 14. Since it does not expect any memory to be present at \$A000, it selects the \$E000-FFFF ROM any time bits 15 & 14 are both 1. This causes an 'image' of the \$E000-FFFF ROM to appear at \$A000-BFFF normally. We will need our own address decoding logic to examine all three of these bits and select between the ROM and our RAM as required.

We use an 74LS138 3-to-8 line decoder to decode the address. Depending on the binary value of its 3 input lines (bits 15,14, & 13 in our case), the decoder selects ONE of its 8 output lines. If the inputs are 000, output line 0 will be selected; if the inputs are 001, output line 1 will be selected; and so on. A selected output line is set low, which allows it to be used directly as a chip select (CS) signal. In the case, we want to select the \$E000-FFFF ROM chip when bits 15,14 & 13 are 111, and select our extra 8K RAM when these bits are 101. We can accomplish this by using bits 15,14,13 as the three inputs to the decoder. Output line 5 (101 in binary) will be connected to the select line of our \$A000-BFFF RAM, and output line 7 (111 in binary) will be connected to the \$E000-FFFF ROM. We'll also have to disable the normal select line to the \$E000-FFFF ROM since we are going to handle the chip selection process with our own decoder.

Refer to figure 1. This figure shows the connections to the 74LS138 address decoder and 8K RAM. The three input lines of the decoder (pins 1-3) are connected directly to the microprocessor's address lines A13-A15. Pins 4-6 can be used to turn the DECODER on and off; for our purposes we want it on all the

time, as connected. Pins 8 and 9 are ground and power (+5v), respectively. The other 8 pins are the output lines. The output lines we need are lines 5 and 7 (pins 10 and 7 respectively). These will be connected to the chip selects of out \$A000-BFFF RAM and \$E000-FFFF ROM, respectively.

The 8K RAM chip will use the address (A0-A12) and data (D0-D7) lines of the \$E000-FFFF ROM, and a couple of additional connections. First of all, the RAM has ground (pin 14) and power (pin 28) connections, 13 address lines (marked A0-A12) and 8 data lines (D0-D7). Pin 27 is the write enable (WE) line, which does not normally run to the ROM. We can pick up this line from the drive's 2K RAM chip, which uses it. Pins 20,22, & 26 are different types of chip select/enable lines for the RAM; suffice it to say that as long as pin 26 is held high (+5v), pins 20 & 22 together can be used to select the RAM by bringing them both low (ground). Thus they are both connected to output line 5 of the 74LS138 decoder, which goes low when the RAM should be selected. Pin 1 has no function.

The 8K RAM chip will be physically mounted on top of the \$E000-FFFF ROM. Rather than risk using the original 24-pin ROM, we will make a copy of it to a 28-pin EPROM. An AD adapter will take care of the translation from the 24-pin socket in the drive to the 28-pin configuration of our new EPROM and the 8K RAM. Almost all of the pins of the replacement EPROM have the same function as the pins of our RAM. After making the connections given above, all that is left is to disable the normal select line for the \$E000-FFFF ROM, since we are using our decoder to select the replacement EPROM. Disabling the normal select line is accomplished very easily by cutting a connection within the AD adapter corresponding to pin 22 of the EPROM.

As you read through the detailed instructions on the next few pages, be sure to refer back to this theory of operation any time you have a question.

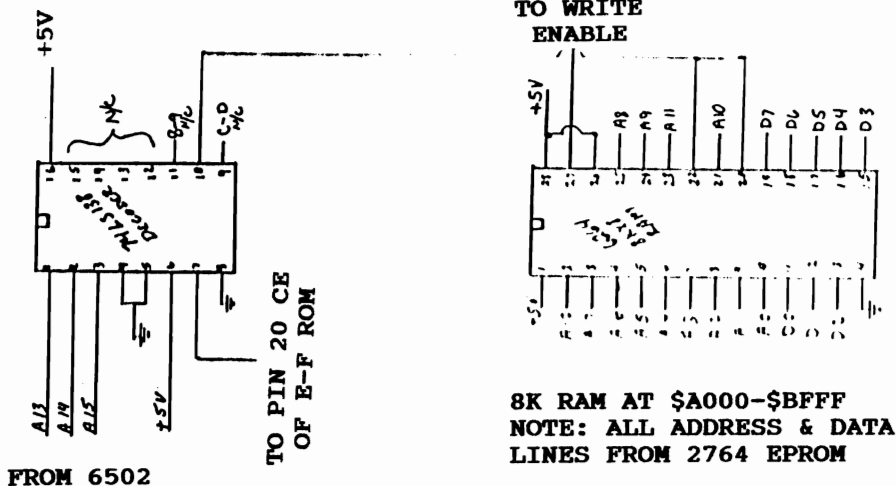


FIGURE 1

IMPORTANT NOTICE:

Read these instructions thoroughly before making any modifications to your disk drive. Any time you modify electronic equipment you run the risk of damaging it. Although this modification has been tested and these instructions have been checked extensively, CSM Software Inc. does not make any warranty, express, implied or otherwise, as to the accuracy of any information contained herein or the performance of your equipment as a result of being modified. You agree to bear the responsibility for any damage or loss that may occur. Under no condition will CSM Software Inc. be responsible for any damage or loss. If you are not comfortable with this situation, DO NOT MAKE ANY MODIFICATIONS TO YOUR EQUIPMENT.

MATERIALS REQUIRED

- > (1) 2764 EPROM (8K)
- > (1) 8Kx8 STATIC RAM (6264,4364,etc.)
- > (1) 74LS138 DECODER
- > (1) AD 24-28 pin adapter
- > (1) 28-pin socket

- > 6' (approx) insulated, solid strand, small diameter (22-30 gauge) wire
- > Low-watt soldering iron (do not use a soldering gun)
- > Resin-core solder (do not use acid-core!)
- > Tape or general-purpose adhesive
- > Access to EPROM programmer

GENERAL NOTES:

- 1) The pins on IC chips are numbered counter-clockwise from notch in one end. Some IC's have a dot by pin 1 instead of a notch. See figure 1.
- 2) In the following procedures, a jumper wire means as short a wire as practical; a long wire means approx. 4-5 inches. Long wires may be trimmed to suitable length before final connection.
- 3) Where the instructions say 'connect' we would use solder. If you don't feel comfortable soldering, you may use wire-wrapping or some other technique.
- 4) Directions such as left, front, etc., are based on looking at the drive as you do in normal use, i.e. from the front.

DISASSEMBLY & ORIENTATION

1. Turn the disk drive upside-down and remove the six screws hold the top of the case on.
2. Turn the drive right-side up. Remove the two small screws which hold the

metal shield on. Remove the metal shield. Note: It may not be possible to reinstall the shield or cover after modifying your drive.

3. Locate the following chips on the circuit board (location varies).

6502: 40-pin chip near center of board (may be labelled C014377).

\$E-F ROM: 24-pin chip labelled 901229, to rear of 6502

\$C-D ROM: 24-pin chip labelled 325302, to left of \$E-F ROM.

2K RAM: 24-pin chip. May be labelled 2116, 2016, 6116, 8128 or other. In all cases it is to the left of \$C-D ROM. Older drives may have four 2114 18-pin chips instead.

ASSEMBLY 1 - 74LS138

1. Connect a jumper wire between pins 4 & 5 of the 74LS138. Connect a jumper from this connection to pin 8. Then connect a long wire to pin 8.
2. Connect a jumper between pins 6 & 16. Connect a long wire to pin 16.
3. Connect long wires to pins 1, 2, 3, 7 & 10.

ASSEMBLY 2 - EPROM/RAM

1. Locate the \$E-F ROM and carefully remove it. This chip is socketed on all drives we've seen. If it isn't socketed on yours, we recommend that you desolder the ROM and install a socket. Since that is quite a job in itself, you may not want to do this modification.
2. We prefer to save the original ROM chip and work with a copy of it instead. Drop the ROM into the EPROM programmer and copy it to the computer's memory, using the directions from 68764-type chips. Remove the ROM and insert the 2764 EPROM. Burn a copy of the ROM into the EPROM. Read the EPROM back out and verify that the copy matches the original. (NOTE: Now's your chance to make modifications to the DOS if desired).
3. Insert the EPROM into the AD adapter, noting that the notched end of the EPROM goes into the end of the AD socket that 'overhangs' the 24-pin header. Plug the AD into the drive temporarily with the notched end towards the rear of the drive, and verify that the drive operates correctly. Unplug the AD from the drive and leave the Eprom in it.
4. Carefully bend out pins 20, 22, 26 & 27 of the 28-pin socket (not the AD). Connect a jumper between pins 20&22 of this socket. Connect a jumper between pins 26&28 (not 27) of this socket. Connect a long wire to pin 27 of this socket.
5. 'Piggyback' the 28-pin socket on top of the EPROM in the AD. Make sure the notched or dotted ends match. Carefully connect all pins of the socket to the

corresponding pins of the EPROM, except for pins 20, 22, 26, & 27 (bent out previously). If soldering, don't use too much heat; permanent damage to the EPROM may result. Insert the 8K RAM chip into the socket, making sure the notched ends match.

FINAL ASSEMBLY

1. Cut the pin in the AD (NOT the RAM socket) between its 24-pin header (bottom) and its 28-pin socket (top) at pin 22 of the SOCKET (which is pin 20 of the 24-pin header). Bend this pin out from the AD socket. Connect the long wire from pin 7 of the 74LS138 decoder to this pin.
2. Connect the long wire from pin 8 of the 74LS138 to pin 14 of the AD socket.
3. Connect the wire from pin 10 of the 74LS138 to pins 20 & 22 of the RAM socket, bent out and jumpered previously.
4. Connect the long wire from pin 16 of the 74LS138 to pins 26 & 28 of the RAM socket, bent out and jumpered previously.

INSTALL IN DRIVE

1. Plug the AD/EPROM/RAM assembly into the socket in the drive where you removed the \$E-F ROM previously. Make sure the notched ends of the chips and sockets all point to the REAR of the disk drive.
2. Using tape or general-purpose adhesive, carefully fasten the 74LS138 chip upside-down to the top of the \$C-D ROM, located next to the AD assembly. Be sure there is no chance of any wires or pins accidentally shorting out anywhere. You may prefer to delay this step until after the testing stage.
3. Connect the long wire from pin 27 of the RAM socket in the AD assembly to pin 21 of the drive's built-in 2K RAM. On older drives with four 2114 chips, connect this wire to pin 10 of any one of the 2114 chips.
4. Connect the wires from pins 1, 2, & 3 of the 74LS138 to pins 23, 24 & 25, respectively, of the 6502. You may prefer to use microclips instead of soldering in these last two steps. If you do use solder, be extremely careful not to apply too much heat, as you may permanently damage your 6502 or 2K RAM. Also be careful not to drop any solder on the drive's circuit board!
5. Double-check all connections. Your modified drive is now ready for testing.

TESTING

Testing the 8K RAM expansion is accomplished using the shore 'TEST RAM' routine given below. A disk monitor such as DRVMON from DI-SECTOR is also required. First, you should type the routine into the computer at \$4300-16. Start by displaying this area of memory with the 'M' command: M 4300 4316

Next, type the following values over the bytes displayed:

```
.:4300 78 A0 00 A9 00 99 00 A0
.:4308 C8 D0 FA EE 07 43 AD 07
.:4310 43 C9 C0 D0 EE 58 60 00
```

Save the program to disk with: S "TEST RAM",08,4300,4317

Now transfer the routine to the disk drive's memory using the 'TC' command:

```
TC 4300 4316 0300
```

Switch DRVMON over to operating inside the drive with:

```
08 (letter 0; digit 8 for device 8)
```

Disassemble the routine with D 0300 and check that it matches this listing:

```
0300 SEI                ;DISABLE INTERRUPTS
0301 LDY #$00
0303 LDA #$00           ;FILLER VALUE
0305 STA $A000,Y        ;$A000 =STARTING LOCATION
0308 INY
0309 BNE $0305
030B INC $0307          ;HI BYTE OF ADDRESS
030E LDA $0307
0311 CMP #$C0           ;STOP AT $C000
0313 BNE $0303
0315 CLI                ;ENABLE INTERRUPTS
0316 RTS
```

Execute the routine using the 'G' command: G0300

The routine will fill drive memory from \$A000-BFFF with the value \$00. When the routine is finished, examine this area with DRVMON to verify that all bytes are now \$00. After performing this test, change the value at \$0304 in drive memory (filler value) to \$55. You must also change the high byte of the starting location (\$0307) back to \$A0. Run the routine again and examine memory. Repeat the test again with the values \$AA and \$FF. This redundancy will assure you that your extra memory is functioning correctly. Your modified drive is now ready for use.

If this testing indicates a problem, reread the instructions and check all your connections again. NOTE: DRVMON 'locks up' occasionally in normal use. It also occasionally inserts a '00,0k,00,00' message into the memory display. This DOES NOT indicate a problem with your 8K RAM memory.

16K MEMORY

Once you've added 8K of RAM to your drive, it is very simple to add ANOTHER 8K of RAM or EPROM for a total of 16K of extra memory. The address decoder installed with your first 8K has the capability of selecting another chip located

at \$8000-9FFF. Recall that chips are selected based on the three highest bits (bits 15,14,13) of the address being accessed. If you look back to the chart on page 2, you'll see that referenced to memory at \$8000-9FFF always have bits 15-13 set to 100, and references to memory at \$C000-DFFF have these bits set to 110. Again bit 14 can be used to distinguish between these two ranges.

In the first 8K extra memory setup, we allowed the normal address decoding logic to select the ROM at \$C000-DFFF whenever necessary. As we noted then, the normal decoder does not use bit 14. This means that it cannot distinguish between references to the \$8000 and \$C000 ranges, so again an image of the \$C000 ROM appears at \$8000. In order to add our second 8K there, we'll have to bypass the normal decoding/selection process and do our own. This can be accomplished by breaking the normal select line to the \$C000 ROM and using our 74LS138 decoder to choose between the two chips as necessary. See figure 1.

ADDITIONAL MATERIALS

In addition to the materials for the first 8K of extra memory, the following parts are required for the second 8K of memory:

Another 8Kx8 static RAM OR Another 2764 EPROM (for EPROM-based routines)

Another 28-pin socket (not in AD) ,(we suggest a ZIF-type if you're adding EPROM)

ADDITIONAL STEPS

To install the second 8K of memory, perform the following additional steps at the indicated points:

ASSEMBLY 1 - 74LS138

Step 4. Connect long wires to pins 9 & 11 of the 74LS138.

ASSEMBLY 2 - EPROM/RAM

Step 6. Bend out pins 20 & 22 ONLY of the second 28-pin socket and jumper them together.

Step 7. Piggyback the second socket onto the FIRST 8K RAM, making sure the notched ends match. Connect all the pins together except pins 20 & 22, bent out previously. Insert the second RAM or EPROM into the socket.

FINAL ASSEMBLY

Step 5. Connect the wire from pin 11 of the 74LS138 to pins 20 & 22 of the SECOND socket, bent out and jumpered previously.

Step 6. Connect the wire from pin 9 of the 74LS138 to pin 20 of the \$C-D ROM (or use a microclip).

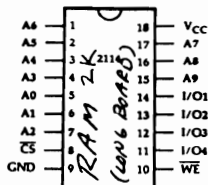
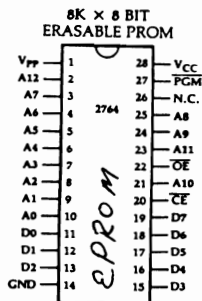
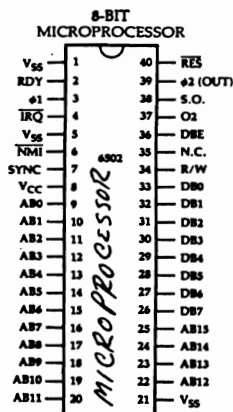
Step 7. Locate the solder 'dot(s)' on the circuit board next to the right side of the \$E-F ROM (on opposite side from the \$C-D ROM). Your drive may have one, two, or three such dots. Each dot has a circuit trace line running from it. Find the dot which has a trace running to pin 20 of the \$C-D ROM. If your drive has more than one dot, the dot closest to the front of the drive will probably be the one you want. You may have to unplug the AD assembly temporarily to verify that you have the correct trace. Using a small screwdriver or knife, VERY CAREFULLY cut completely through this trace, near the solder dot. Be sure to cut AWAY from any other solder traces to avoid damaging them. Make sure the trace is completely cut. If you ever wish to return your drive to its original state, you will have to connect a small jumper wire across the cut in this trace.

Note: The chip in the bottom socket is at \$A000-BFFF and the chip in the top socket is at \$8000-9FFF. If you would like this to be the other way around, simply swap the wires coming from pins 10 and 11 of the 74LS138.

TESTING

In the 'TEST RAM' routine, use \$80 as the high byte of the starting address (\$0307) if you are adding another 8K of RAM. If adding EPROM instead, use an EPROM with know contents (erased EPROM's contain all \$FF's). In either case, examine memory from \$8000 on to verify correct operation.

VARIOUS PINOUTS



TRACK READ

TRACK READ will read an entire track of GCR data from the disk into the 8K expansion RAM at \$A000-BFFF. The routine works by waiting for a sync mark and then reading the next 8K of GCR data into memory. This is enough GCR data to assure that the longest track possible on the 1541 will be read completely. On shorter tracks, the first part of the GCR data will be duplicated at the end. To use TRACK READ, load and run DRVMON. Switch DRVMON over to the drive with '08' and type in the program at \$0300 using the assembler. Transfer the program to the computer with TD 4300 4360 0300 and save it to disk with S "TRACK READ",08,4300,4360. Execute the program by storing the desired track no. at location \$0006 and storing \$E0 at \$0000. The program stores a status code at \$0000 when it is done; a \$01 code means OK and a \$03 means no sync was found on the track.

```

0300 78      SEI
0301 A9 00   LDA #$00      ;SET POINTER ($14-15) TO
0303 85 14   STA $14      ; TO START OF BUFFER ($A000-BFFF)
0305 A9 A0   LDA #$A0
0307 85 15   STA $15
0309 A2 20   LDX #$20      ;32 PAGES TO FILL
030B A0 00   LDY #$00
030D A9 00   LDA #$00      ;FILL BUFFER WITH $00'S
030F 91 14   STA ($14),Y
0311 C8      INY
0312 D0 FB   BNE $030F
0314 E6 15   INC $15      ;HIGH BYTE BUFFER POINTER
0316 CA      DEX
0317 D0 F6   BNE $030F
0319 20 00 FE JSR $FE00    ;SET PCR TO READ MORE
031C A9 A0   LDA #$A0      ;START BUFFER POINTER AT $A000
031E 85 15   STA $15
0320 A2 00   LDX #$00      ;SET UP TIMER FOR SYNC
0322 A0 00   LDY #$00
0324 88      DEY           ;COUNTDOWN TIMER AND
0325 D0 07   BNE $032E    ; CHECK FOR SYNC
0327 CA      DEX
0328 D0 04   BNE $032E
032A A9 03   LDA #$03      ;ERROR - NO SYNC
032C D0 1F   BNE $034D
032E 2C 00 1C BIT $1C00    ;CHECK FOR SYNC
0331 30 F1   BMI $0324    ;BRANCH BACK IF NO SYNC
0333 AD 01 1C LDA $1C01    ;SKIP FIRST BYTE
0336 B8      CLV
0337 A0 00   LDY #$00
0339 A2 20   LDX #$20      ;32 PAGES TO READ
033B 50 FE   BVC $033B    ;WAIT FOR BYTE READY
033D B8      CLV
033E AD 01 1C LDA $1C01    ;LOAD BYTE FROM DATA PORT
0341 91 14   STA ($14),Y  ;STORE DATA IN BUFFER

```

0343	C8	INY	
0344	D0 F5	BNE \$033B	
0346	E6 15	INC \$15	
0348	CA	DEX	
0349	D0 F0	BNE \$033B	
034B	A9 01	LDA #\$01	;01=NO ERROR
034D	85 00	STA \$00	;STORE ERROR CODE IN COMMAND QUEUE
034F	4C 6E F9	JMP \$F96E	;ROM ROUTINE TO END

HARDWARE LIMITATIONS OF THE 1541 DRIVE

We have seen many different program protection schemes used on the 1541 drive. We have even developed many of the protection schemes that are currently being used by a number of different software companies. We have researched the 1541 disk drive and found its physical limits. We know what can and what can not be duplicated on the 1541 disk drive. There are many different protection techniques that may be detected by the 1541, yet these same techniques are not able to be produced on the 1541. This fact is due to the physical limitations of the disk drive. More and more we are seeing protection schemes that capitalize on the hardware limits of the disk drive. These hardware limits will be the controlling factors as to whether or not the end user will be able to copy a disk or not.

Our business is program protection. We protect programs for other companies and we protect our own. We also teach people the how's and why's of program protection. We are not some pompous company that claims to have the 'TOP SECRET STUFF' on program protection (only to give you something less than you bargained for). We take the time to investigate many different protection schemes and most importantly, we share the knowledge that we have learned. It is through this time-consuming and laborious investigation process that we have gained our knowledge. Now on to the topic at hand.

Let's take a look at the various options that a programmer must consider when choosing a viable and effective program protection scheme.

First and foremost, there are no secrets to program protection. Each program is protected through a logical and definable set of instructions. Each instruction will have a specific purpose and cause a specific task to be performed. The programmer is able to control which task is to be performed and in what order it is to be performed.

Second, the technique that the programmer chooses must be something that is capable of being detected on the 1541 disk drive (i.e. nybble count, track arcing, synchronized tracks, extra tracks, modified formats etc.). It is important to note that the protection scheme chosen is one that must be detectable on the 1541 drive. It is not necessary or even desireable to have the scheme producible by the 1541 drive.

Third, the protection scheme chosen should not be easily reproduced on the 1541. If possible the scheme selected should not be able to be written on the 1541 under any circumstances. In other words, the scheme selected should take advantage of the hardware limitations of the 1541 drive.

The hardware limits of the drive include, but are not limited to, the following:

1. The small amount of RAM available in the 1541 drive (2K total). In order to accurately reproduce a track, large amounts of memory are sometimes required (e.g. data blocks longer than 1K).
2. The width of the R/W head on the 1541 is wide in comparison to what is required. This prevents the use of adjacent tracks and half-tracks.
3. The drive itself uses 4 separate densities during the normal course of operation. This allows the use of mixed densities on a single track, which is very difficult to copy.
4. Speed variations from disk to disk and from track to track on the same disk. This is a hinderance to nybble counting and nybble copying.
5. Speed variations also make track synchronization, track arcing and spiral tracking difficult to reproduce. As the speed varies it causes the relationship between tracks (and half-tracks) to vary.
6. The 1541 does not reference the timing hole on the disk. This compounds the problems of speed variation when moving the R/W head over large distances.

Let's now examine each of the hardware limits presented here in a little more detail. We will view each limitation as to how it applies the making a copy of the original disk.

1. LIMITED RAM... Approximately 1K of the 1541's RAM is required for internal housekeeping (stack, 0 page pointers, communications, etc.) and for the ML program used by a copy program to read from and write to the disk. This only leaves us with apx. 1K for a data storage buffer. When we read data in from the disk it is necessary to temporarily store it the drive's RAM memory before it is transferred over to the computer. If the original disk uses data blocks longer than 1K, most copy programs will have an extremely difficult time copying the disk.

When reading data from a disk the limited RAM is not a great hindrance. It is possible to sync up to a particular reference point, then to read 1K of memory and transfer it to the computer. Then all the copy program has to do is find this same reference point on the next revolution of the disk, skip over the first 1K and then read the second 1K of data from the disk. This process may be repeated until the whole track has been transferred to the computer's memory.

Writing data back out to the disk is where the real problem comes into play. It is necessary to write data out in 1K chunks (or less) due to the limited RAM in the drive. Data must be written to the disk in 'sync'. This means that the data must be timed out to the disk in a predictable and reliable pattern. The only way to maintain this predictable pattern is to write the data out in one continuous stream, with no interruptions. The way that the 1541 overcomes this problem of synchronizing the data is by keeping the total data block sector length to 330 GCR bytes (plus or minus). Each time that the data block is written to the disk, the sync mark and the data block identifier are also written out. This insures that the data will always be synchronized with its sync mark.

If we should write out 1K of data and then try to write out additional data in synchronization with the first chunk of data, there will be problems. The timing relationship of the drive and the disk prevent the data from maintaining the perfect relationship they had on the original disk. If the timing varies, even by one bit, the data of the second chunk will now be unintelligible and the copy will fail. In practice the data will be found to have varied by more than just a few bits and by as many a dozen bytes from the proper orientation. Rotational speed of the disk also affects how well the data will match the original. Disks vary in rotational speed, from track to track and from disk to disk. The speed also varies during a single revolution of the disk. This variation is not predictable, nor is it reproducible. The variance of speed plays an important part in how accurately the two chunks of memory match on the copy disk. In reality it is difficult, if not impossible, to maintain the relationship between 2 or more blocks of data written to the disk.

2. WIDTH OF THE R/W HEAD... Tracks on the 1541 drive are spaced at 48 tracks per inch (apx 0.020 in.). There are many commercial disk drives that use a track spacing of 96 tpi (0.010 in. apx). Since the tracks are further apart on the 1541 each track may be wider than tracks written on the 96 tpi drive. Due to the width of the R/W head of the 1541 drive it is not possible to write adjacent tracks and half tracks side by side.

If an original disk is prepared on a drive that uses the 96 tpi drive, it is possible to write adjacent tracks and half tracks. Each track will be narrower than the standard 1541 track, but they will be readable if done properly.

3. DENSITIES... The 1541 drive uses 4 different densities to write data on every disk. The density refers to the time between bits on the disk. The outer tracks will time bits out at a higher rate than the inner tracks. The higher density allows the 1541 drive to store more data on the outer tracks. Normally, density is set to a specific value for a given track and this density is not changed on the track. If the programmer should change density in the middle of the track it would be next to impossible to detect and reproduce the density changes. If the programmer used multiple density changes and combined this with various other techniques the difficulty level would be tremendously increased.

4. SPEED VARIATIONS OF THE DRIVE... As seen above, speed variations of the drive can be a hinderance to copy programs. Speed will have a direct effect upon the total number of bytes written to the track, the density of the bytes written and the relationship between sectors on a track. If the speed varies by just 1% on a track this could throw off the nybble count by as much as 77 bytes. This means that the total track length could be 77 bytes longer or 77 bytes shorter than the original track. During the normal course of operation the drive will compensate for the variation in speed from disk to disk by varying the length of the gap bytes. In a nybble count of a track it is not important which bytes are gap bytes or which are data bytes; they are all counted. This assumes that no other changes have been made to the track such as density or extra long data blocks. (note: it becomes even more difficult to copy the disk when multiple schemes are employed).
5. SYNCHRONIZED TRACKS... Each track has a beginning and an ending. If we use sector 0 as the beginning and sector 20 as the end of a track we can begin to establish our track to track relationship. The 1541 drive writes data out to the disk during the format process in a somewhat random procedure. This means that each and every track could start any where on the disk in relation to the timing hole (we are using the timing hole only as a handy reference point on the disk - the 1541 does not use the timing hole for any purpose whatsoever). If the original disk was created on a programmable drive, each and every track could be synchronized to the timing hole, thereby insuring the proper synchronization from track to track.

While it is possible to time each track and establish the relationship from track to track and then synchronize one track to another, it gets very difficult to synchronize multiple tracks. Track arcing and spiral tracking are variations of synchronized tracks whereby relatively small amounts of data are written to the disk on adjacent tracks and half-tracks. The reason that track arcing and spiral tracking works is due to the fact that at no time is the data on a track and its adjacent half-track side by side. For instance, if the programmer wrote 1/4 of a track and then stepped the head a half track and continued writing data for another 1/4 of the track the information would be staggered on the disk. This staggering of data is what allows these schemes to function properly.

The next time that you try to copy one of your 'latest and greatest' programs and you find that there is not any copy utilities that will copy them, you won't have to wonder why.
Copy protection has reached the physical limits of 1541 drive.

What about hardware-modifying the drive to overcome some of these limitations of the 1541?? Well, this has been tried, mostly without much success (see the product review of the TRACKMIMIC(TM) elsewhere in this issue). The limiting factor here is that 1541 has so many hardware limitations that it is not practical to try to overcome all of them.

IMPORTANT NOTICE.....

Folks, if you buy one of our competitor's products and are not happy with it, DON'T complain to us. Unfortunately, we can not help you when you buy what seems to be a similar product from another company. During the past few months we have received many complaints from people about other cartridge backup systems. There has been a tremendous surge in companies trying to jump on the cartridge backup band wagon (four different companies that we know of). Don't be misled by confusing claims and advertising. Check out your purchases carefully. If you are not sure of something, ask specific questions.

BIT COPYING ON THE 1541 (PART 1)

One of the questions that comes up again and again is why can't the 1541 make an exact bit-for-bit copy of a track. Since the 1541 can read and write whole sectors at a time, why can't we just extend the process and read whole tracks? It seems like such a simple thing to do, yet it's basically impossible on the 1541. Even if a 1541 drive created the original disk, it doesn't mean a 1541 can copy it. The following is the first of a two-part series to try and put the whole situation in perspective and settle the issue once and for all. To understand this discussion, it is also important to read the chapters on GCR recording and the standard disk format in PROGRAM PROTECTION MANUAL II.

Let's start by examining the process of reading bits on the 1541. It may surprise you to learn that the only thing a 1541 can detect DIRECTLY is a "1" bit. A "0" bit can only be detected INDIRECTLY, by the ABSENCE of a "1" bit. You can think of a bit as a tiny magnet with a north pole (N) and a south pole (S), so it looks like this: NS. The magnet is always lined up parallel to the track it's on, but it can still be put on the track two different ways: NS or SN. Now, you might think the 1541 would use one way for a "0" bit and the other way for a "1" bit, but this is NOT the case. Rather than detect the ACTUAL DIRECTION of the magnet, the 1541 can only detect a CHANGE in the direction. In other words, it can only tell when a magnet is placed differently than the one before it, like NS SN or SN NS. In this case the second magnet will be interpreted as a "1" bit.

If two magnets are placed the same way, like NS NS, they actually form a single magnet, twice as long (just like toy magnets do). The 1541 will not detect any change when it comes to the second magnet; it will interpret this ABSENCE of a change as representing a "0" bit. How does the 1541 know when the second magnet was SUPPOSED to start, so it can tell that no change took place at that point? It's all a matter of timing. On tracks 1-17, for instance, bits are allotted a maximum of 3.25 microseconds (millionths of a second, or us). Suppose the drive has just read a "1" bit. It knows exactly WHEN the "1" bit occurred because it detected the change in magnetic field. After the "1" bit has passed, the drive knows the next bit should occur within 3.25 us. If no change in the magnetic field occurs within this time, it will assume the next bit is a "0".

For this precise timing scheme to work, the diskette must be spinning at precisely the correct rate: 300 rpm. This equals 5 revolutions per second, or 200,000 us per revolution. Suppose the 1541 has just read a "1" bit on track 1,

and there is another "1" bit coming up next. If the diskette is spinning too slowly, the "1" bit might not appear within the allotted 3.25 us. The drive would incorrectly conclude the next bit was a "0" since it did not see a change within the allotted time. An extra "0" bit would be inserted in the data. Likewise, suppose the next two bits are a "0" and a "1", in that order. If the diskette is spinning too fast, the "1" bit could come around within the 3.25 us that was allotted for the "0" bit. The drive would conclude the next bit was a "1", and so it would miss the "0" bit.

Variations in speed can (and do) occur no matter how well the drive's speed is adjusted. Due to the drag of the diskette and other factors, the speed will always vary slightly as the disk is turning. Two provisions are made to get around this problem. The first provision has already been mentioned: the timing is always based on when the last "1" bit occurred (since this can be detected precisely). This is taken care of by the drive's hardware and cannot be changed by software.

If there are too many "0" bits in a row, a variation in speed can easily throw off the timing. How many "0"'s in a row is too many? Well, the designers of the 1541 decided to never put more than two "0"'s in a row on the disk, to be safe. To enforce this, normal data bytes are converted into a special form called GCR code before being written on the disk. GCR code has been carefully designed so that no possible combination of GCR bytes will ever result in more than two "0" bits in a row. The conversion to GCR is done by the DOS software in the 1541, however. This means you can get around it by controlling the write circuitry with your own routines. You can WRITE as many "0"'s in a row as you want, but you'll have trouble reading them reliably, i.e. you'll have trouble making a bit-for-bit copy.

Even worse than too many "0"'s is mixing different bit-densities on a track. We said above that bits on tracks 1-17 are allotted 3.25 us each. On other tracks, the timing is different (because the physical sizes of the tracks are different). There are four zones on a 1541 diskette. Each zone consists of a set of tracks: tracks 1-17, 18-24, 25-30 and 31-35, respectively. Within a particular zone, the tracks all use the same bit timing (density) and all have the same number of sectors. The different timing factors are given in the following table:

ZONE NO.	RANGE OF TRACKS	BIT-RATE: BITS PER SEC	BIT-TIME: TIME PER BIT	BYTE-TIME: TIME PER BYTE	SECTORS PER TRACK	MAX BYTES PER TRACK
1	1-17	307,692	3.25 us	26 us	21	7692
2	18-25	285,714	3.5	28	19	7142
3	26-30	266,666	3.75	30	18	6666
4	31-35	250,000	4.0	32	17	6250

Using two different densities on a single track will cause effects similar to speed variation. Suppose we're writing bits to the disk at zone 1 density (3.25 us per bit) and then we suddenly switch to zone 4 density (4.0 us) and write some more. Now suppose we try to read the track using 3.25 us timing. When we come to the place where the density was switched, the bits will suddenly start

coming along "slower". The drive may conclude a bit was a "0" because no change in the magnetic field happened within the allotted 3.25 us, even though if it had waited the full 4.0 us it would have detected a change. The combination of mixed density and the inevitable speed variations of the diskette will make this track extremely difficult or impossible to copy bit-for-bit.

A protection scheme, however, can still check for the correct data on the track if it knows where the density changes happen. For instance, the density may change right after a certain pattern of bits occurs. When the drive detects this pattern, the protection routine can immediately switch densities. Reading at the new (abnormal) density, it will look for another specific pattern. If it doesn't find the pattern it wants, the disk is not an original. The protection scheme can take appropriate steps to crash the program. By putting multiple changes of density on a single track, a very secure protection scheme is created.

Other factors besides mixed density and speed variations also combine to make it impossible to copy a track exactly. These factors were covered in the article "Hardware Limitations of the 1541" in the April 85 newsletter. They include such things as the limited RAM in the drive (2K), the relatively slow speed of the processor (1 MHz) and the lack of a index hole sensor. Next month we'll look at these factors again, and also take a look at what kind of hardware we'd have to have in order to even come close to a bit-for-bit copier.

BIT COPYING ON THE 1541 (PART 2)

Last month we gave you some insights into the technical background of bit copiers. Just to refresh your memory, let's cover the highlights:

1. Data is stored on the disk as a series of magnetic "domains". A domain is a single magnet (NS) or consecutive series of magnets (NSNSNSNS). The domain ends and a new domain begins where LIKE poles are placed consecutively (NS SN or SNSN NS SN). The order of the magnetic field (NS or SN) is not important, only change in the field by the adjacent like poles.
2. The disk drive actually depends upon the magnetic domains to change direction within a specific time limit.
3. Due to fluctuations in the drive's rotational speed, successive "0" bits in a row can greatly affect the ability of the drive to correctly read the data from the disk. Remember, a "0" bit is NO change in magnetic domains within a specific time limit and a "1" bit is a change in magnetic domain within the specific time limit.
4. There are four densities at which the 1541 can time (clock) bits by.

I hope everyone is familiar enough with the basics to get on with our subject (if not, be sure to review the relevant chapters in PROGRAM PROTECTION MANUAL Vol. II). All disk drives (including our 1541) rely upon a very

fundamental rules of operation: Was there a change in the magnetic domain (magnetic field) within an allotted time period? If a change occurred within the time limit, a "1" bit is registered. If NO CHANGE occurred within the allotted time limit, a "0" bit is registered. The time period is referred to as the density.

Now let's get on to one of the reasons that the 1541 will not be able to copy every disk, every time. Last month we established that the 1541 has four separate densities at which bytes are written to the disk. While the 1541 is not physically capable of changing the density of individual bits as they are written, it is possible to accomplish this task with more sophisticated equipment. Actually, it is possible to change the density of each and every byte (or bit) on the disk to any density. While it may seem quite extreme that anyone would go to such great lengths to protect a disk, it is certainly possible. The more practical use of density changes is to write a small number of bytes to the disk, change the density after a predetermined number of bytes and do this hundreds of times on a single track. As long as the programmer knows where and when to change density it would be a simple process to write a routine that will switch density so that the special track may be identified.

The problem comes in when the bit copier tries to read the data from the disk. Only one density may be selected at any one time. This means that some of the bytes may be read at the wrong density and give false readings. Also, if the programmer is nybble counting the bytes on the track, the copy disk will fail due to multiple density changes that are required to match the number of bytes on the original disk. It is easy to see that just by varying the density it is possible to give most nybble copiers fits.

How is it possible that some of the more sophisticated disk duplicators can copy any disk?? They have a tremendous amount of onboard RAM and they use many different microprocessors that run at terrific speeds, instead of using just one microprocessor to perform all the tasks in the drive (like the 1541). By reducing the disk down to the fundamental level, it is possible to read and reproduce every conceivable bit pattern. At the fundamental level we are only concerned with the time that has lapsed between magnetic domains. In order to accomplish this task it would require a machine capable of detecting a change of domain and recording the time to the next change. Can we add this ability to the 1541, to enable it to reproduce a track bit-for-bit? What else would we need to add to the 1541? Well, that's like asking what you would have to add to a Volkswagen to make it into a Cadillac! Basically, you throw away the Volkswagen and build the Cadillac from scratch! Okay, so what hardware do we need to build into our Cadillac of disk duplicators, so it can handle ANY combination of density changes, even from bit to bit?

Let's start with the clock speed. We have to be able to distinguish between a bit that is 3.25 us long, for instance, and one that is 3.50 us long, so we have to be able to detect a difference of 0.25 us. This will require at least a 4 Mhz clock speed (4 million "ticks" per second), since we can only measure a WHOLE NUMBER of clock ticks. With a 4 Mhz clock, 3.25 us would be 13 ticks, while 3.50 us would be 14 ticks. Thus, the difference will be a whole clock tick. Actually,

we would probably want an even faster clock, say 6 or even 8 Mhz, to give us a extra margin of reliability.

Next, let's consider how much RAM memory would we have to use in order to store a whole track. If the whole track was written at the same density, we'd only need 8K of RAM (see PPM Vol. II). However, we'll need much more RAM to handle cases such as switching between all four densities, 15 to 20 times on a single track. To store a "picture" of the track in memory, each bit in RAM will represent one CLOCK TICK (not one bit of disk data). The memory will be filled with "0" bits initially. Each time our clock ticks, we'll check the disk to see if a change in the magnetic domain occurred. If no change occurred, we'll leave the current RAM bit a "0", otherwise we'll change the current RAM bit to a "1" to mark the change. This means we need 1 RAM bit for each clock tick. Since it takes 1/5 sec for the disk to rotate once, and the clock ticks 4 million times a second, we need 800,000 RAM bits to store ONE track. Now 800,000 bits = 100,000 bytes = about 100K bytes of RAM memory to store ONE track! With overhead such as storage for the duplication program itself, we can figure we'll need 128K of RAM.

Once we have the ability to duplicate a whole track, we still won't necessarily be able to copy a whole disk. We also have to consider track synchronization, that is, where the beginning of the tracks are relative to each other. If you picture a clock face superimposed on the disk, track 1 might start at 12 o'clock but track 2 might start at 3 o'clock. A protection scheme can check for this. In order to absolutely insure that each track begins where it's supposed to, we need an index hole sensor. This will allow us to detect each time the index hole located near the hub of the disk comes around. We can measure the start of each track from the index hole and thus line up the tracks correctly.

All in all, reproducing a disk exactly turns out to be a lot more complicated than you might have thought! We may return to this subject again in the Newsletter after you've had time to digest the information we've presented in these articles.

1541 FREEZE UTILITY (For 8K RAM)

This month we present the 1541 FREEZE UTILITY, a very useful aid for investigating program protection within the 1541 drive or debugging your own custom DOS routines. The Freeze utility allows you to interrupt the disk drive at any time. The entire RAM memory of the drive will be preserved for you to examine. All of the 6502's registers will be preserved too, so you can tell exactly what the processor was doing when you interrupted it. The Freeze utility also has a special feature which allows you to restore the memory and registers, and resume execution at the point where it was interrupted! This means you can make changes to the routine and test them. You can even change the contents of the registers or start execution at a different point. Those of you that work with program protection and custom DOS routines will appreciate all the possibilities this opens up.

In order to use the Freeze utility, you must install an extra 8K of RAM memory in your 1541. Instructions for the extra memory are provided with the EXTRA RAM KIT available from CSM, which includes an 8K RAM and an address decoder chip. In addition, you'll need a 2764 EPROM, a 24/28-pin adapter, a 28-pin socket and access to an EPROM programmer such as the PROMENADE. All of these supplies are available separately from CSM. To install the Freeze utility, follow the instructions for the extra RAM. Instead of using an exact copy of the \$E000-FF (E-F) DOS ROM, however, you must insert the ML code on the next 2 pages into the ROM. Start by putting an exact copy of the ROM in C64 memory at \$2000-3FFF. Then assemble the first sections of code at the locations indicated in the \$2000-3FFF area. Next, flip out the BASIC ROM (change location \$0001 to a \$36). Then assemble the rest of the code at \$AF00 as indicated (for convenience). Finally, move the \$AF00 code down to the ROM copy using T AF00 AF95 3F30. Save the ROM copy to disk, then burn it onto EPROM.

When the drive is first turned on, a special routine will be installed in the extra RAM memory at \$AF00. You may then use the drive as usual. When you want to interrupt the drive, you must cause an NMI interrupt (see below). All of normal RAM memory (\$0000-07FF) will be copied up to the extra RAM starting at \$A000. Then the drive will be RESET, which does not affect the extra RAM. Then you may examine the memory copy or transfer it to the C64, using a disk monitor such as DRVMON or STARMON (from DI-SECTOR V2 and V3 respectively). You may use the drive to load in your disk monitor or other utilities; the extra RAM will not be disturbed. You can determine the location at which the routine was executing from the program counter (PC). This information is stored on the stack, along with the A, X, and Y registers and the processor status (P). The stack pointer (SP), which is stored at \$A800, indicates where you may find these values. Starting with location \$A100 + SP + 1, the next six bytes are the Y, X, A, P and PC (lo/hi) registers, respectively. The bytes on the stack after the PC often indicate the return address (minus 1) for subroutine calls that led the program to the point indicated by the PC. Once the 1541 memory has been copied up to the extra RAM via an NMI interrupt, a second NMI will restore the memory to the normal RAM and restart execution! By altering the PC value on the stack, you can make it start executing anywhere.

To generate an NMI interrupt, you must briefly ground the NMI input of the 6502. We use a logic probe for this, but you can get by with just a piece of wire. Connect one end of the wire to the "-" terminal on one of the large cylindrical capacitors on the rear of the board. Then briefly touch the other end to pin 6 of the 6502. A switch could also be used.

1541 FREEZE PROGRAM LISTING

ASSEMBLE THIS CODE INTO A COPY OF THE E-F DOS ROM AT \$2000 IN THE C64

```
;2780 60      RTS          Disable utility loader.
;2781 78      SEI          RESET entry point
;2782 D8      CLD
;2783 A0 04    LDY #$04
;2785 88      DEY
```

```

;2786 F0 15      BEQ $279D
;2788 B9 00 AF    LDA $AF00,Y    Check if routine already
;278B D9 30 FF    CMP $FF30,Y    copied to RAM (check
;278E F0 F5      BEQ $2785      "CSM" identifier).
;2790 A0 00      LDY #000        If routine not there already,
;2792 B9 30 FF    LDA $FF30,Y    copy from ROM
;2795 99 00 AF    STA $AF00,Y    to extra RAM.
;2798 C8          INY
;2799 C0 B6      CPY #$B6
;279B D0 F5      BNE $2792
;279D 4C A0 EA    JMP $EAA0      Continue normal RESET.
;27A0 EA          NOP            Fill in rest of utility
;27A1 EA          NOP            loader routine area.
;27A2 60          RTS

:2AE4 EA EA      Disable ROM checksums
:2AE8 EA EA      (insert NOP's)

:3FFA 04 AF      New vectors: NMI ($AF04)
:3FFC 81 E7      and RESET ($E781).

```

ASSEMBLE THIS CODE UNDER THE BASIC ROM IN THE C64, THEN MOVE IT TO \$3F30

```

:AF00 00          Up/download flag (0=upload).
:AF01 43 53 4D    "CSM" Identifier.
:AF04 78          SEI
:AF05 48          PHA            Save A,
:AF06 8A          TXA
:AF07 48          PHA            X
:AF08 98          TYA
:AF09 48          PHA            & Y.
:AF0A BA          TSX
:AF0B BD 04 01    LDA $0104,X    Check if called from "UI" command
:AF0E C9 6E      CMP #$6E        (return address = $CB6E).
:AF10 D0 0E      BNE $AF20      If not, branch to freeze routine.
:AF12 BD 05 01    LDA $0105,X
:AF15 C9 CB      CMP #$CB
:AF17 D0 07      BNE $AF20      " " " " " "
:AF19 68          PLA            If "UI", remove registers from
:AF1A 68          PLA            the stack ...
:AF1B 68          PLA
:AF1C 58          CLI
:AF1D 4C 01 FF    JMP $FF01      ... and jump to normal "UI".
:AF20 A9 03      LDA #$03        MAIN FREEZE ROUTINE
:AF22 A2 00      LDX #$00        Debounce NMI line
:AF24 A0 00      LDY #$00        (wait approx. 1 sec.)
:AF26 CA          DEX
:AF27 D0 FD      BNE $AF26
:AF29 88          DEY

```

;AF2A D0 FA	BNE \$AF26	
;AF2C E9 01	SBC #\$01	
;AF2E D0 F6	BNE \$AF26	
;AF30 BA	TSX	
;AF31 BD 06 01	LDA \$0106,X	Check if extra NMI's happened.
;AF34 C9 AF	CMP \$AF	
;AF36 D0 09	BNE \$AF41	
;AF38 68	PLA	If so, pull last NMI info
;AF39 68	PLA	from stack
;AF3A 68	PLA	
;AF3B 68	PLA	
;AF3C 68	PLA	
;AF3D 68	PLA	
;AF3E 4C 30 AF	JMP \$AF30	... and check again.
;AF41 AD 00 AF	LDA \$AF00	Flip up/download flag.
;AF44 49 FF	EOR #\$FF	
;AF46 8D 00 AF	STA \$AF00	
;AF49 F0 24	BEQ \$AF6F	Branch if was set for download.
;AF4B BA	TSX	UPLOAD memory.
;AF4C 8E 00 A8	STX \$A800	Save stack pointer.
;AF4F A9 07	LDA #\$07	
;AF51 8D 5D AF	STA \$AF5D	Start source at \$0700.
;AF54 A9 A7	LDA #\$A7	
;AF56 8D 60 AF	STA \$AF60	Start destination at \$A700.
;AF59 A0 00	LDY #\$00	Copy one page
;AF5B B9 00 07	LDA \$0700,Y	from source area
;AF5E 99 00 A7	STA \$A700,Y	to destination area.
;AF61 C8	INY	
;AF62 D0 F7	BNE \$AF5B	
;AF64 CE 60 AF	DEC \$AF60	Decrement destination page no.
;AF67 CE 5D AF	DEC \$AF5D	Decrement source page no.
;AF6A 10 EF	BPL \$AF5B	If source >= page 0, more to do.
;AF6C 4C A0 EA	JMP \$EAA0	Done - jump to normal RESET.
;AF6F A9 A7	LDA #\$A7	DOWNLOAD memory.
;AF71 8D 7D AF	STA \$AF7D	Start source at \$A700.
;AF74 A9 07	LDA #\$07	
;AF76 8D 80 AF	STA \$AF80	Start destination at \$0700.
;AF79 A0 00	LDY #\$00	Copy one page
;AF7B B9 00 A7	LDA \$A700,Y	from source
;AF7E 99 00 07	STA \$0700,Y	to destination.
;AF81 C8	INY	
;AF82 D0 F7	BNE \$AF7B	
;AF84 CE 7D AF	DEC \$AF7D	Decrement source page no.
;AF87 CE 80 AF	DEC \$AF80	Decrement destination page no.
;AF8A 10 EF	BPL \$AF7B	If dest. page >= 0, more to do.
;AF8C AE 00 A8	LDX \$A800	DONE - retrieve stack pointer.
;AF8F 9A	TXS	
;AF90 68	PLA	Restore Y,
;AF91 A8	TAY	
;AF92 68	PLA	X

;AF93 AA
;AF94 68
;AF95 40

TAX
PLA
RTI

& A
Return from original NMI interrupt.

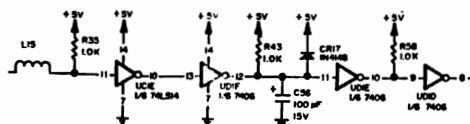
1541 RESET SWITCH

By now everybody has seen instructions on how to install a RESET switch in the C64. Usually these switches are installed so that they RESET the entire serial bus, including the computer, drive, printer, etc. This is the simplest way to do things, and it is adequate for most situations. Occasionally, however, you may want to RESET the disk drive separately from the computer. For example, with SNAPSHOT 64 and similar utilities it's often necessary to turn the disk drive off and back on after the original program has loaded, before you save the snapshot to disk. Turning computer equipment on and off should be avoided as much as possible, so it would be helpful to be able to RESET the drive without affecting the computer. Also, some drives will actually RESET the computer by themselves when you turn them off and back on, so a drive RESET switch can be a necessity. Finally, a drive-only RESET switch can be useful when you're investigating protection schemes or working on custom DOS routines.

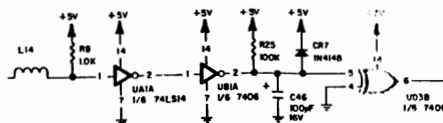
It's easy to install such a switch, and the cost is minimal. You'll need a normally open / momentary contact push button switch, some light wire and two microclips. Start by attaching a sufficient length of wire to both contacts of the push button switch. Attach the clips to other ends of the wires. Now you must locate the proper points in the drive to clip onto. Depending on whether your drive has the long or short circuit board, the points are different. In long board drives the circuit board covers the hub of the disk, whereas in short board drives it doesn't. In either case, one clip should be attached to the metal frame of the drive as a ground (it doesn't matter which clip).

The second clip has to be attached to a particular chip on the circuit board. The chips are labelled differently on the two types of boards (see the diagrams below). On the long board, clip to pin 11 of UD1E. On the short board, clip to pin 5 of UD3B. Be sure the metal part of the clip doesn't make contact with any other components. Turn the drive's power on and test the switch by pushing it momentarily. The drive will RESET but the computer will not. Attach the switch to the case, reassemble the drive and you're done.

LONG BOARD



SHORT BOARD



With this setup, RESEtting the computer will still RESET the drive too. If you want to prevent this from happening, you'll have to cut a wire on the board. For the long boards, cut one wire on L15 (it doesn't matter which one). For the short boards, cut a wire on L14.

1571 REVISIONS

Well, there's good news and bad news (maybe). Commodore is reportedly preparing a new revision of the ROM for the 1571 drives (1571's currently on the market use ROM revision 3). The new ROM, revision 4, is undergoing in-house testing at this time. So far there's no real information about when the new ROM might become available or whether there'll be any upgrade policy for current 1571 owners. As far as the changes planned, the big news is that Commodore has apparently fixed the notorious SAVE @ bug, thanks in part to Dr. Phil Slaymaker, who wrote Peek-A-Byte. The SAVE @ bug plagues most of Commodore's disk drive models, including the 1571 (see the Nov. 1985 Newsletter). The fix may not be complete, however; it may only reduce the chances of a SAVE @ problem. Revision 4 will also fix some other minor bugs.

An even bigger change in the 1571 may be in the works. Commodore is reportedly redesigning the circuitry in the 1571. Current 1571's use a Western Digital 1770 drive controller chip to handle the MFM (non-Commodore CP/M) disk formats. The chip will be replaced by a new custom chip manufactured by Commodore. Commodore has a history of manufacturing all their own chips, and the change should cut costs. When they replace the MFM chip, they'll also have to change the MFM routines in the ROM, resulting in revision 5. This revision will include all of the revision 4 changes too. Another change they may make in revision 4 or 5 has to do with the process used to detect a disk's type of format. In 1571 mode, the drive can take 10-20 seconds to "recognize" a 1541 format disk the first time you read it after it's inserted. If you didn't know any better, you might think your drive or disk was out of alignment. This format detect process may be streamlined.

Another hardware change is also rumored. Current 1571's contain a 6526 interface chip (CIA) like the ones used in the C64 and C128 to handle most peripherals. The CIA in the 1571, which handles the fast serial and "burst" communication modes, may be eliminated too. Whether the CIA's functions will be taken over by some new chip, or by the 6522 (VIA) chips already used in the 1571 (and 1541) drives, is not certain at this time. Eliminating the CIA could cause compatibility problems with current 1571 software that uses the burst mode. Copy programs specifically for the 1571 may be the most seriously affected. Although Commodore's source code for the 1571 ROM contains warnings to third-party developers that both these chips will "go away", many developers didn't receive the source code or ignored the warnings.

While we're on the subject of rumored revisions, one caller reports that they bought a 1541 drive and discovered that it has an optical track 1 sensor

(like the 1571). As you probably know, up to now the 1541 has used the dreaded "bump" (physical stop) to determine the track 1 position, causing 99% of drive alignment problems in the process. The Newtronics (flip-lever) 1541's do have a mounting hole and arm for an optical sensor. If the "bump" has been eliminated, this would be big news. It may also involve a change in the DOS, which would be the first since late 1983, and raise questions about compatibility. We haven't seen one of these new 1541's ourselves, so we can't confirm they exist. Anyone out there have more info? In a final note, the 1572 dual drive is apparently dead. Despite recent advertisements for it, Commodore is not working on a 1572 currently, and apparently has no plans for it in the future. Too bad - seems like a lot of people were interested in it.

ALTERNATE KERNAL DISK ROUTINES

In this article we'll look at some alternate ways of communicating with the 1541 disk drive that are not commonly known. Because of this, they can be used as a form of protection. Communicating with the drive is normally accomplished using subroutines in the KERNAL which are accessed through a jump table (a set of JMP statements). For instance, the normal sequence of events to send a command to the drive goes like this. First, we use the KERNAL routines SETLFS (\$FFBA) and SETNAM (\$FFBD) to set up certain parameters such as the device number. Then we use OPEN (\$FFC0) to open a logical file. A logical file is a communications channel, and should not be confused with the physical files on a diskette. To send a command or data to the drive through a logical file, the file must first be set for output using CHKOUT (\$FFC9). The information is sent using CHROUT (\$FFD2). To receive a response, you set up the file for input with CHKIN (\$FFC6) and read in the data with CHRIN (\$FFCF). Finally, you use CLOSE (\$FFC3) to signal the end of communication.

Note that the addresses of all of these routines begin with \$FF, since the KERNAL jump table is located at \$FF81-FFF5. When you're looking for a protection routine in a program, you can often find it just by looking for these \$FF bytes. To mislead people, a programmer might avoid using the jump table and call on the KERNAL routines directly. For instance, when you call the OPEN routine at \$FFC0, the first thing it does is JMP to \$F34A, where the OPEN routine is actually located. So a clever programmer might use JSR \$F34A instead of JSR \$FFC0. The bad news is that there is a slight risk involved in doing this. Commodore promises that the OPEN routine can always be reached through JSR \$FFC0, but it doesn't guarantee that the OPEN routine will always be located at \$F34A. In a future revision of the C64 KERNAL, it might be moved to a different location. The good news is that in all the KERNAL revisions so far, no routines have been moved around (with one minor exception, CINT). See 'KERNAL ROM REVISIONS' in the May newsletter).

Bypassing the jump table is a good technique, but we can do much better. We can also do without the common KERNAL disk routines entirely! OPEN and its related routines are provided for convenience. OPEN stores the logical file number, device number and secondary address (disk channel) parameters for a file

in a special FILE TABLE so that they don't have to be specified each time. The CHKOUT and CHKIN routines look up these parameters and use them to establish an output or input channel with the correct device. To do this they make use of some other standard but less common KERNAL routines. LISTEN (\$FFB1), SECOND (\$FF93), CIOUT (\$FFA8) and UNLSN (\$FFAE) are used to send output from the C64 to the drive. TALK (\$FFB4), TKSA (\$FF96), ACPIR (\$FFA5) and UNTLK (\$FFAB) are used to receive input from the drive. These routines are usually accessed through the KERNAL jump table (notice all the \$FF's), but we can also call them directly.

How do you use these routines? To send information to a drive you must first tell the correct device to pay attention by using the LISTEN routine. Next, the channel number must be sent as a secondary address with the SECOND routine (all communications to the drive must go through one of the drive's channels, which are like the computer's logical files). You then use CIOUT to actually send the bytes of information. When you are finished sending information, you must let the drive know by using UNLSN (unlisten). Receiving information from the drive is a similar process. The TALK routine tells the correct device to get ready to send. The TKSA (talk secondary address) routine specifies which channel to use (you can't use SECOND to do this). The ACPIR routine receives the bytes of information, and the UNTLK (untalk) routine tells the drive to 'shut up'.

Sounds pretty simple, doesn't it? It is, but there are a couple of complications. The first time a channel is used, the drive must be commanded to 'open' it (reserve a buffer for the channel). This is done by going through the LISTEN/SECOND/UNLSN process, adding \$F0 to the channel number before sending it with SECOND. Once a channel has been opened and UNLSN'ed, it can be reopened by adding \$60 to the channel number and calling SECOND or TKSA. When a channel is no longer needed, the drive should be notified so it can 'close' the channel (release the channel's buffer for other use). Not doing this can cause problems with SAVE @! A channel is closed by going through the LISTEN/SECOND/UNLSN process, adding \$E0 to the channel number.

Let's look at these routines in action. They can be used to send any type of command or data to the disk, such as U1, M-E, M-W, B-E, etc. (for that matter, they can be used to communicate to ANY serial bus device, such as a printer). Suppose you want to check for a 'bad block' error on a disk. The most common way, as you probably know, is to send a 'U1' command to the drive and then check the error channel. A routine to do this is given in the PROGRAM PROTECTION MANUAL Vol. I (subroutine #1, chapter 11). Get out your copy if you have one and look over the routine to refresh your memory. Similar routines are used in many, many commercial programs. The routine below checks for a bad block too, but it uses LISTEN, TALK, etc. rather than OPEN, CHKOUT, etc. It also bypasses the jump table and calls the routines directly. The result is a routine that does not 'look' like a protection routine at all, even to an experienced eye. Even if you concluded that this was a protection routine, you would still have to delve into the KERNAL code to figure out what it does. This would be considerably easier with a book such as ANATOMY OF THE C64, but still not simple.

Here's how the routine works. The first thing we do is clear the I/O status byte (location \$90), which corresponds to the BASIC system variable ST. Then we

tell device 8 to listen, so we can tell it to 'open' channel 15 (\$FF=F0+0F). As you know, channel 15 is the error/command channel. Next, we 'unlisten' channel 15 and open channel 2. We want to make channel 2 a random access channel, so we use CIOUT to send a '#' character (this is like OPEN 2,8,2,"#"). Now we unlisten channel 2 and 'reopen' channel 15 (\$6F=60+0F) so we can send the "U1" command. Using a loop, we send the command bytes to the drive with CIOUT. When the command has been sent, we must unlisten channel 15. The command will not be executed by the drive until we do. Once the drive performs the command, it will produce an error (or ok) message through the error channel. We want to read in this error message, so we tell channel 15 to TALK. Another loop is set up to receive the message bytes with ACPIR and store them in memory. Note the use of the I/O status byte to check for the end of information (EOI, which Commodore calls End Or Identify). Bit 6 of the status is set to 1 when the last byte is received. The status byte can also be used to check for conditions like 'device not present' after a LISTEN or TALK command. Once EOI is signalled, we untalk channel 15. Since we're done with them, we close channels 2 and 15 (note that \$E0 is added to the channel numbers). In a real application, we would go on and check that the correct error was received, etc.

With a little practice you'll become familiar with these 'new' routines. Perhaps you'll want to go even further. Get yourself a book such as ANATOMY OF THE C64 and study the interrelations between the various serial bus routines. Once you've untangled them, you could incorporate their algorithms into your own program and avoid the KERNAL entirely!

1000	A9 00	LDA #\$00	
1002	85 90	STA \$90	Clear I/O status
1004	A9 08	LDA #\$08	Device number
1006	20 0C ED	JSR \$ED0C	Command dev. 8 to LISTEN
1009	A9 FF	LDA #\$FF	Open channel \$0F (\$F0+0F)
100B	20 B9 ED	JSR \$EDB9	Send SECOND address
100E	20 FE ED	JSR \$EDFE	Command dev. to UNLISTEN
1011	A9 08	LDA #\$08	
1013	20 0C ED	JSR \$ED0C	LISTEN, dev. 8
1016	A9 F2	LDA #\$F2	Open channel \$02 (\$F0+02)
1018	20 B9 ED	JSR \$EDB9	Send SECOND address
101B	A9 23	LDA #\$23	'#' (random file)
101D	20 DD ED	JSR \$EDDD	CIOUT - send '#' byte
1020	20 FE ED	JSR \$EDFE	UNLISTEN
1023	A9 08	LDA #\$08	
1025	20 0C ED	JSR \$ED0C	LISTEN
1028	A9 6F	LDA #\$6F	Reopen channel \$0F (\$60+0F)
102A	20 B9 ED	JSR \$EDB9	SECOND
102D	A0 00	LDY #\$00	Reset index
102F	B9 80 10	LDA \$1080,Y	Get byte of 'U1' command
1032	F0 06	BEQ \$103A	Quit if byte = \$00
1034	20 DD ED	JSR \$EDDD	Otherwise, send byte
1037	C8	INY	Increment index
1038	D0 F5	BNE \$102F	Continue if more
103A	20 FE ED	JSR \$EDFE	UNLISTEN

103D	A9 08	LDA #\$08	Device number
103F	20 09 ED	JSR \$E009	Command dev. 8 to TALK
1042	A9 6F	LDA #\$6F	Reopen channel \$0F (+ \$60)
1044	20 C7 ED	JSR \$EDC7	TKSA - send TALK sec. addr.
1047	A0 00	LDY #\$00	Reset index
1049	20 13 EE	JSR \$EE13	ACPTR - get byte from disk
104C	99 90 10	STA \$1090,Y	Save in memory
104F	A5 90	LDA \$90	Get I/O status
1051	29 40	AND #\$40	Pull out bit 6 (EOI)
1053	D0 03	BNE \$1058	Quit if end of information
1055	C8	INY	Otherwise, increment index
1056	D0 F1	BNE \$1049	Continue if more
1058	20 EF ED	JSR \$EDEF	Command device to UNTALK
105B	A9 08	LDA #\$08	
105D	20 0C ED	JSR \$ED0C	LISTEN
1060	A9 E2	LDA #\$E2	Close channel \$02 (\$E0+02)
1062	20 B9 ED	JSR \$EDB9	SECOND
1065	20 FE ED	JSR \$EDFE	UNLISTEN
1068	A9 08	LDA #\$08	
106A	20 0C ED	JSR \$ED0C	LISTEN
106D	A9 EF	LDA #\$EF	Close channel \$0F (\$E0+0F)
106F	20 B9 ED	JSR \$EDB9	SECOND
1072	20 FE ED	JSR \$EDFE	UNLISTEN
1075	00	BRK	

1080	55 31 3A 35 20 30 20 33 35 20 31 36 00	U1:5 0 35 16
------	--	--------------

&-AMPERSAND FILES

Among the most mysterious and little known features of the 1541 DOS are AMPERSAND FILES. These files are not documented in the 1541 User's Guide that comes with the drive, but books such as INSIDE COMMODORE DOS have revealed their existence by uncovering the DOS routines which process them. INSIDE COMMODORE DOS calls these files "UTILITY LOADER FILES", which is apparently also what Commodore calls them. We prefer to use the term "AMPERSAND FILES" (&-FILES) since in most cases the name of the file involved must begin with an ampersand character ("&"). We'll also use the term &-ROUTINE for the DOS routine found in an &-FILE.

&-ROUTINES are relatives of the BLOCK-EXECUTE (B-E) routines discussed in EXECUTING 1541 ROUTINES in the June newsletter. Recall that the B-E command loads an entire block off the disk into one of the drive's 256-byte buffers, located at \$0300, \$0400 etc. The code in the buffer is then executed starting at the first byte. An &-routine, on the other hand, can load into any location in the drive's RAM and can be any length up to 256 bytes. The &-routine will be automatically executed starting at the first byte loaded. Amazingly, a single &-FILE can contain any number of separate &-ROUTINES, and each &-routine can be automatically loaded into a different area of memory in any order! Only the first &-routine loaded will be automatically executed. Considering their potential in

program protection, the rarity of &-files is probably due to a lack of documentation. At least one well-known program uses them in its protection scheme: DI-SECTOR V2.0. We'll come back to the DI-SECTOR &-file, but first let's take a look at a simpler example.

&-FILES have an internal structure different from any other type of Commodore file. First of all, they must be USR-type files. There isn't anything mysterious about USR files themselves; they are linked together just like program or sequential files. In fact, you can use a USR file just like a PRG or SEQ file if you specify type "U" in your LOAD, SAVE or OPEN statement. For instance, to save a program into a USR file, use SAVE"0:PROG,U",8. To open a USR file for sequential writing, use OPEN 2,8,2 "0:DATA,U,W". This makes it easy to create an &-file. Besides being of type USR, &-files have a special internal format:

FORMAT:	LOAD	CODE				
	ADDRESS	LENGTH	ML CODE	CHECKSUM	
EXAMPLE:	00 03	09	A9 29 85 77 A9 49 85 78 60		2D	

The first two bytes of an &-file (after the track/sector link) make up the load address, in standard lo-byte/hi-byte format. In the example above, the load address is \$0300 (\$00 03). The next byte is the code length byte, specifying how long the &-routine's ML code is, NOT counting the load address, code length or checksum bytes. The ML code above is 9 bytes long and it follows immediately after the code length byte. Following the ML code is the checksum byte, which is calculated in a special way. All the bytes of the &-routine are ADDED together, INCLUDING the load address, code length byte and the ML code itself. Any time the total goes over \$FF a carry is produced, which is added back into the checksum. Note that this is NOT the same technique used to calculate header and data block checksums when writing sectors to the disk. In fact, the "&" checksum corresponds to what is called 2's-complement addition, if that rings any bells. There is a subroutine in the DOS at \$E84B to add another byte into a checksum, but rather than look at it we'll present a more useful subroutine you can use to checksum your own &-routines in the computer.

CHECKSUM &-FILE

1000	A0 00	LDY #000	Zero out Y-index ...
1002	84 FF	STY \$FF	... and checksum
1004	18	CLC	Start main loop - clear carry
1005	B1 FB	LDA (\$FB),Y	Get next byte into accumulator
1007	65 FF	ADC \$FF	Add current checksum
1009	69 00	ADC #000	Add carry (if any)
100B	85 FF	STA \$FF	Save as new checksum
100D	A5 FB	LDA \$FB	Compare low bytes of current
100F	C5 FD	CMP \$FD	and ending addresses
1011	D0 06	BNE \$1019	Branch if more to do
1013	A5 FC	LDA \$FC	Compare high bytes ...
1015	C5 FE	CMP \$FE	
1017	F0 08	BEQ \$1021	Branch if done
1019	E6 FB	INC \$FB	Increment low byte of current address ...

```

101B D0 E7 BNE $1004
101D E6 FC INC $FC      ... and high byte if necessary
101F D0 E3 BNE $1004    Loop back
1021 00 BRK             Return to ML monitor

```

To use this routine, put the starting address of the memory area to be checksummed into \$FB-C in standard lo/hi byte format. Put the ending address of the area to be checksummed into \$FD-E. Execute the routine from an ML monitor with: G 1000. The checksum will be in location \$FF after the routine finishes. The checksum is actually calculated by the code at \$1004-0E. After clearing the previous carry, the next byte to be added to the checksum is loaded into the accumulator. The ADC (Add with Carry) statement at \$1007 then adds together three things: the carry (just cleared), the accumulator (containing the next byte) and the current checksum from location \$FF. The low byte of the result is placed back into the accumulator. If the result was over \$FF, the carry is set to 1; otherwise it is cleared to 0. To add the carry back into the checksum, we perform another ADC at \$1009. This adds the carry, the accumulator and the value \$00 together (\$00 is just a dummy value). The new checksum is then stored back into \$FF. The rest of the routine checks the ending address and loops back if there is more code to be checksummed.

Type in the checksum routine from the monitor. Now let's use this routine to verify the checksum in our &-file example. Enter the bytes shown in our example into memory at \$2000, using the monitor's M command. The &-file looks like this in memory, minus the checksum:

```

:2000 00 03 09 A9 29 85 77 A9
:2008 49 85 78 60

```

The area to be checksummed is from \$2000-200B, so put these addresses in \$FB-C and \$FD-E:

```

:00FB 00 20 0B 20

```

Execute the routine with G 1000. When it is finished, examine location \$FF for the checksum. It should be \$2D. Put this checksum into location \$200C, right after the last byte of code (\$60). Now our &-file is ready to write out to the disk. The easiest way to do this is with a short BASIC program. Exit the monitor, type "NEW", and enter the following program:

```

10 OPEN 2,8,2, "0:&9,U,W" :REM Drive 0, file &9, type USR, Write mode
20 FOR I=8192 TO 8204      :REM Save area from $2000 to $200C
30 PRINT#2, CHR$(PEEK(I)); :REM Get byte from memory and write to file
40 NEXT I                  :REM Loop till done
50 CLOSE 2                 :REM Close file

```

This program simply writes out the memory from \$2000-200C to a USR file called "&9". This program is set up for our &-file example; change the locations in line 20 for routines located elsewhere in memory.

The &-file we have just created will change the drive's device number to 9 when it is executed. The ML routine to do this is very simple:

```
2003 A9 29 LDA #$29 Device number (9) OR'ed with $20
2005 85 77 STA $77 Save as LISTEN address
2007 A9 49 LDA #$49 Device number (9) OR'ed with $40
2009 85 78 STA $78 Save as TALK address
200B 60 RTS Return from &-routine
```

No need to worry about HOW this changes the device number; just accept that it does. In fact, it's the machine language equivalent of the BASIC method given in the drive manual. Note that we started our code at \$2003 to leave room for the load address and code length bytes, and that the &-routine ends with an RTS.

OK, now that we've created our &-file, how do we execute it? There are two ways: a software method (&-command) and a hardware method (boot clip). The software method involves sending the name of the &-file as a command to the drive, through the command channel (15). The file name MUST begin with an "&" to use this method, so it's often called the &-command. Our routine can be executed very easily with the following command:

```
OPEN 15,8,15,"&9"
```

(Note that the command given in INSIDE COMMODORE DOS, p. 362 is not correct). Execute the "&9" file now. If all goes well, the drive will be changed to device 9. If the checksum was not correct, you will get an error 50, 'RECORD NOT PRESENT'. If the code length byte is too large, the DOS will reach end of file prematurely and you'll get an error 51, 'OVERFLOW IN RECORD'. Note that both of these errors are normally associated only with relative files!

The hardware method for executing an &-file involves the CLOCK and DATA lines of the serial bus. If these lines are held low (grounded) with a BOOT CLIP (piece of wire. etc.) before the drive is turned on, it will wait for the lines to go high (+5V) and then load and execute the FIRST file on the disk. In this case only, the file name does not have to start with an "&". The boot clip feature was removed from the latest revision of the DOS (rev 5), and probably won't be missed.

As mentioned above, a single &-file can contain any number of &-routines. Simply start the load address of the second &-routine right after the checksum of the first &-routine, and so on. The first file called "&-----&" on DI-SECTOR V2.0 uses this technique. It contains three &-routines, starting at byte \$02 of sector 18/5, byte \$1D of 18/5 and byte \$23 of 18/6. The second &-routine has a \$00 for the code length byte, which actually specifies 256 bytes of code. Take a look at this file if you have DI-SECTOR V2.0. Now that you know how &-files work, experiment!

THE COMMODORE 128 COMPUTER

The new Commodore 128 computer seems to be catching on. A lot of readers we've talked to either want to know more about the C128 or have already bought one. We've had a couple C128's (and 1571 drives) here for several months now, and we've been working with them as time permits. As you might expect, the C128 is a bigger machine than the C64 in more ways than one, so we won't try to cover the whole thing in one article.

For starters, of course, the C128 has a 64 mode that is almost completely compatible with C64 software. Then there's CP/M mode. For now, all we need to say about this mode is that if you've seen one CP/M system, you've seen 'em all (almost). In fact, the CP/M reference books offered for sale in the C128 manual contain NOT ONE WORD about the C128 machine in particular (or any other particular machine, for that matter). This brings us to the third mode of the C128, namely 128 mode.

The 128 mode is like an expanded and upgraded C64 in many ways. People have compared it to having a C64 with a BASIC programming utility, graphics expander, music program, ML monitor and fast load utility all running at the same time. Plus 128K of RAM and a 2MHz mode for nearly twice the speed of the C64. Plus a new 80 column RGB video chip (8563), many new screen editing commands, a new memory management chip (MMU), a much better power supply, a RESET button, etc. Nicer keyboard, too. The keys look great and feel great. The numeric keypad should be real handy, as well as the four separate cursor keys and display pause button. The C64 really needed a lot of those things all along, and now the C128's got them. It's a great new system in the same spirit as the C64. The C128 is similar enough to the C64 that it's easy to get comfortable with it, but it has a lot of new features to keep you busy exploring for quite a while.

The C128 looks good "under the hood" too, inside the machine. The circuit board looks neat and tidy. It's encased in metal shielding all around, which cuts down on radio interference quite a bit compared to the C64 (you can thank the FCC for that). The shielding also doubles as a heat sink for the main chips, so the machine hardly gets warm at all. While there have definitely been some defective machines around (we got one), in general the C128 seems to be very well-made piece of equipment (ditto for the 1571 drive). Even the plastic case is more solid, more attractive and more functional than the C64's.

Inside the machine in a different way, the C128 KERNAL ROM is not just a patched up version of the C64. Although some parts are similar, of course, it's all been reorganized and rewritten completely. The new bank-switching chip (MMU) used to switch the system ROMs in and out of memory is much more flexible and organized than on the C64. Contrary to popular belief (it seems), the C128 DOES allow cartridges to be used; in fact they can have up to 32K of memory, twice as much as cartridges on the C64. Besides the regular "external" cartridges, there's an empty socket inside the machine for an "internal" cartridge (32K EPROM). Provision is also made for a DMA RAM expansion cartridge on the C128 for up to 1 megabyte of RAM memory.

All in all, a great new machine. The new programs that have come out for it so far seem to be very good, too. While an IBM PC or PC compatible still seems to

be the way to go for serious business applications, the C128 could turn out to be the new home computer standard, like the C64 it in its day. For anyone who feels like they're finally outgrowing their C64 now, the C128 is definitely the logical next step.

CUSTOMIZING THE KERNAL

Everybody has their own list of changes they'd like to make to their Commodore 64. Maybe you don't like the screen colors, or you're tired of typing ",8" every time you load a program, or you'd like the screen to say 'WELCOME TO BOB'S COMPUTER' when you turn the computer on. These and other changes can be accomplished easily. You'll need an EPROM programmer, a 2764 EPROM and a 24/28-pin adapter. A reference book such as 'The Anatomy of the Commodore 64' is also immensely helpful, especially if you want to go beyond the modifications we present here.

To start off, you should get a copy of revision 3 of the KERNAL if you don't already have it (SX-64 owners can use their present KERNAL). You could proceed by copying the KERNAL down to RAM memory at, say, \$4000-5FFF, but then you'll have to translate addresses back and forth. A better way is to copy the KERNAL ROM down to the RAM underneath it and switch out the ROM. You can run your system from the RAM copy as long as you don't try to modify a system routine as it is being executed (such as the screen editor and IRQ routines). This way you'll be able to test your changes as you make them. You'll have to copy the BASIC ROM down to RAM as well, since BASIC gets switched off with the KERNAL. You won't be able to use a cartridge based monitor either, since cartridges are also switched out. Of course, you should save your custom KERNAL to disk from time to time as a precaution, but this method works well for us.

Start by copying BASIC and the KERNAL down to RAM with T A000 BFFF A000 (BASIC) and T E000 FF E000 (KERNAL). Now switch off the ROMs by changing location \$0001 to a \$35. You shouldn't notice any change, but you're now operating out of RAM. Now change location \$FDD6 to \$E5 (be sure and change it back to \$E7 before burning an EPROM copy!). This change prevents the RESET routine from switching the ROMs back in when you do a 'soft' RESET (SYS 64738 or G FCE2). Since many modifications will change the startup defaults, you'll need a soft RESET to test them. A 'hard' RESET with a RESET button will still switch the ROMs back in, but it won't wipe out your RAM copy. Try a soft RESET now with G FCE2. After an extra-long pause you will get the familiar startup screen, only with 51216 bytes free! Relax, this just means the RAM test routine got all the way to the VIC control register at \$D011 before finding any 'non-RAM'.

The table below lists some short patches you can make, just for starters. Let's take a brief look at each of them, starting with everyone's favorite, the default colors. The default border and background colors are located in a table of values that get written into the VIC chip on powerup, RESET or RUN/STOP-RESTORE by the VIC initialization routine at \$E5AA. The border and background color codes are located at ECD9-DA, respectively. These colors are

specified using the color RAM codes (\$00=black, \$01=white, etc.) NOT the ASCII print commands (\$90=black etc.). The default character color is located at \$E535, in the screen initialization routine (which also calls the VIC chip initialize routine). The character color is also specified using color RAM codes.

Next on our list is a patch to automatically enable repeating for all keys. To do this we'll have to insert a JSR in the screen initialize routine and put a short piece of code in the unused area at \$E4B7-D2. This simply stores an \$80 into the key repeat flag and returns. While you're at it, you might want to adjust the delay before a key repeats and the speed at which it repeats (note two different locations to change speed). The new values given are just suggestions, so experiment.

Now we come to a real handy pair of changes. First, we modify the BASIC LOAD and SAVE commands so they automatically go to disk rather than tape when no device is specified (i.e. LOAD "PROG"). If you need to specify a secondary address, though, you'll still have to type the device number (LOAD "PROG",8,1). Likewise, you can default the OPEN command to go to the printer if no device is specified. You'll still have to supply the device number if you need a secondary address. Neither of these changes affect the corresponding KERNAL routines, since you must always specify the device number for them (unless you want to get really tricky).

Our next modification is to change the SHIFT-RUN/STOP (Sh-R/S) combination so that it loads the first program off the disk and runs it. To do this, you must first change the default LOAD device as given above. The Sh-R/S combination works by putting its command into the keyboard buffer and then jumping to the keyboard interpret loop. Since the keyboard buffer is only 10 characters long, this limits our possible modifications. To fit in a LOAD ":@" (return) RUN (return) command, we have to use the shifted shortcuts, e.g. R shift-U for RUN, shown as rU. The only way to get around this limitation is to patch into the keyboard interpret loop at \$E5EE, and have it perform your custom command directly.

Our final modification involves the startup screen, which is printed in three separate parts by a routine at \$E422-46. The first part, up to '64K RAM SYSTEM', is stored as a table of ASCII characters at \$E473-AB. The number of bytes free is calculated and printed next, and then the 'BASIC BYTES FREE' message at \$E45F-72 is printed. The first part printed can be altered by just changing the ASCII codes. To disable the bytes free and reuse its message area, make the last two modifications shown. You can then continue your screen message in the 'BASIC BYTES FREE' area.

If you want to go much beyond the modifications here, you'll probably need some extra space to put code in. The only unused area left is at \$E4B7-D2, but you could free up some space (nearly 2K!) if you don't need the cassette. Once you've made your changes, burn the code into a 2764 EPROM and install it in the computer using the AD adapter. Powerup and take your new computer for a test drive!

<u>LOCATION</u>	<u>ORIGINAL</u>	<u>MODIFIED</u>	<u>DESCRIPTION OF MODIFICATION</u>
-----------------	-----------------	-----------------	------------------------------------

FDD6	E7	E5 see text	CHANGE TO \$E7 BEFORE BURNING EPROM!
ECD9-DA	0E 06	Your choice	Default border, background colors
E535	0E	Your choice	Default character color
E536-8	STA \$0286	JSR \$E4B7	Patch to key repeat routine
E4B7-BF	AA ...	STA \$0286	Save character color
	LDA #\$80	Load value - repeat all keys
	... AA	STA \$028A	Save in key repeat flag
	(filler)	RTS	Return to screen init routine
E53A/EB1D	04	03	Key repeat speed - change 2 places
EAEA	10	08	Key repeat delay
E1DA	01 (tape)	08 (disk)	Change LOAD/SAVE default device
E228	01 (tape)	04 (printer)	Change OPEN default device
E5EF	09	0A	No. chars in SHIFT-RUN/STOP command
E5F4-F5	E6 EC	Bf E4	Location of " " " "
E4C0-C9	AA ... AA	4C CF 22 3A	New version " " " "
		2A 22 0D	10":*" (return)
		52 D5 0D	rU (return)
E473-AB	"**** COM.."	Your choice	Change startup message (see text)
E45F-72	"BYTES FREE"	Your choice	Reuse bytes free space
E430-32	LDA \$37; SEC	JMP \$E43D	Disable no. bytes free calc/print

DON'T USE NYBBLE COPIERS TO BACKUP UN-PROTECTED SOFTWARE!!

If you have a nybble copier (or two) DON'T, I repeat DON'T, use them to back up your unprotected software or any disk that you wish to write to!! In order for you to fully understand why we recommend that you don't use the nybble copiers we would like you to review a little information from the PROGRAM PROTECTION MANUAL VOLUME II. Specifically the chapters entitled: STANDARD 1541 FORMAT and CUSTOM DOS ROUTINES (especially EXTRA SECTORS).

Now that you are all familiar with the 1541 format let's get down to business. ZAXXON was the first program to use the technique of adding an extra sector on a track as a means of protection. This was really a pretty neat trick. When they created the disk, they used a special disk drive that was capable of writing an extra sector on tracks 18-24. Many of the programmable disk duplicating machines can handle this scheme with ease. The 1541 disk drive was not designed with these extra sectors in mind.

What we are faced with here is trying to squeeze an extra sector on a track. We say squeezed because there is, quite frankly, not enough room for the sector on the track. So what some nybble copiers do is squeeze the data closer together by reducing the number of header gap bytes, shortening the length of the sync marks and reducing the inter-sector gap bytes. The room that was freed up by squeezing the data closer together will be used by the extra sector. By freeing up some space these nybble copiers are able to accurately copy disks that use extra sectors.

The problem comes when you use these nybble copiers to copy a disk with only the normal amount of sectors. While they are perfectly capable of making a good copy of the disk, the trouble appears when you write to the disk. The DOS (Disk Operating System) is designed to write to the disk based upon specific timing constants (length of header gaps and sync byte lengths). Let's look at what happens during the normal operation of the disk drive during a write operation:

- 1) The drive searches for the particular header. The drive reads all the headers, one by one, and compares them to the desired header. The drive continues until it finds the desired one.
- 2) Once the drive finds the desired header, the drive switches from the read mode (that it used to find the proper sector) to the write mode (so that it may write out the data block). But the problem occurs before the switch is made. DOS waits for the 8 gap bytes to pass before turning on the write mode. Remember that the normal 1541 disk will have 8 gap bytes, but the modified may only have a few gap bytes. Once the DOS has counted 8 bytes, the write mode will be enabled and the data block written out, starting with its sync mark. This wait period is critical. It ensures that the new data block sync will be written out directly over the existing data block sync.

As you can see, the shortened header gap bytes can cause the appearance of two data block sync marks on this sector. The original sync put there by the nybble copier and the new data block sync put there by DOS. When this happens the drive will return a #22 READ ERROR (data block not present). DOS will only consider the first sync mark found as a valid sync. The second will be ignored.

The other problem that occurs is when the nybble copiers just shorten sync byte lengths and inter-sector gap lengths. Here the data block will be written out correctly, but the chance exists for the data block to overwrite following header sync mark and/or the header itself. In this case you will find an #21 or a #20 read error.

Remember DON'T use nybble copiers to copy your un-protected disks or any disk that you may wish to write to in the future!!! Instead use a copy program that is six, or more, months old. When backing up your unprotected software we recommend: CLONE, SPEED COPY (GERMAN COPY), OMNI-CLONE, BACKUP 228, 1541 BACKUP, etc. These programs maintain the integrity of the header, the header gap bytes, sync bytes and the inter-sector gap bytes. In other words, they make a copy that is 'normal' according to 1541 specifications.

EXECUTING 1541 ROUTINES

As program protection evolves, schemes involving bad blocks checked with a UI command are becoming less common (finally). The newer schemes often use custom DOS routines executed inside the disk drive's memory. There are several ways to accomplish this, including MEMORY-EXECUTE (M-E), BLOCK-EXECUTE (B-E), ampersand files (&:) and the job queue commands JUMP (\$D0) and EXECUTE (\$E0). Ampersand

files are very rarely used, so we won't consider them at this time. We'll start our discussion with M-E.

M-E is used to execute a routine which is already in the drive's memory. The routine can either be in the drive's 2K of RAM or in its 16K of ROM. The only requirements are that you know where the routine is located and that the routine end with an RTS. Books such as THE ANATOMY OF THE 1541 DISK DRIVE and INSIDE COMMODORE DOS will help you to find useful routines in the DOS ROMs. In the case of a RAM routine, you have to put the routine in memory yourself before you can execute it. There are a couple of different ways you might do this.

The first way is to use a MEMORY-WRITE (M-W) command. This command can be used from either BASIC or machine language. You must open the command channel first (e.g. OPEN 15,8,15) and then send the M-W command through this channel. The general form of the M-W command in BASIC is:

```
PRINT#15,"M-W";CHR$(lo);CHR$(hi);CHR$(n);CHR$(d1);CHR$(d2); ...
```

'Lo' and 'hi' specify the low and high bytes of the address in the drive's memory to start putting the data in, 'n' is the number of bytes you will send in this M-W command (up to a maximum of 34) and 'd1', 'd2' etc. are the data bytes themselves. You must send all this information (except the "M-W" itself) in CHR\$() form, not ASCII. For instance, if you want to send 5 bytes, you must specify this with CHR\$(5), NOT "5" (which is the same as CHR\$(53)). Also, you should use semicolons (;) as punctuation in BASIC, not commas. The drive expects each byte of information to be in a fixed position in the command and the extra spaces inserted by a comma will throw it off. Most important of all, DO NOT put a colon (:) after the "M-W" or use "MEMORY-WRITE" instead of "M-W". The older disk manuals and many reference books show these being used but they don't work. This has been corrected in the newest Commodore manuals.

Another way to get a routine into 1541 RAM is to load it directly from diskette using either the BLOCK-READ (B-R) or USER1 (U1) commands. The U1 command is by far the most common way, since B-R has several bugs in it. When using U1, special care must be taken to ensure that the block is loaded into the correct memory location. You must first open the command channel to the disk and then open up a # channel too. A typical example of opening a # channel is :

```
OPEN 1,8,2,""
```

This form lets the disk drive select which of its buffers to load the block into. The drive has five 256-byte buffers available, from buffer 0 at \$0300-03FF on up to buffer 4 at \$0700-07FF. Since many ML routines must reside in a particular place in memory, and since you have to know where a routine is to execute it with M-E, you need to be able to select a specific buffer to be used. Fortunately, you can control this by simply specifying the buffer number after the "#", e.g. OPEN 1,8,2,"#0". Having opened the # channel, you are ready to send the U1 command. The syntax for U1 is :

```
PRINT#15,"U1: ";channel;drive;track;sector
```

'Channel' is the channel specified in the secondary address of the OPEN command (2 in the above example), NOT the buffer number (0). 'Drive' must be 0 on the 1541. In contrast to the M-W (and M-E) commands, the information in a U1 command must be sent as ASCII characters. This can be done several ways. For example, a 5 could be sent as using the string "5", the number 5 or the variable V (assuming V = 5). It could even be sent as CHR\$(53), the ASCII value of "5", BUT NOT AS CHR\$(5). You may use a colon after the U1 but it's not required. You may use a space or almost any character as long as there is SOMETHING there. For punctuation you should use semicolons, although spaces are acceptable within strings. The following are valid U1 commands:

```
PRINT#15,"U1:";2;0;1;0
PRINT#15,"U1:2 0 1 0"
```

Once the routine has been read in with U1, sent via M-W or located in ROM, you are ready to execute it with an M-E command. This is actually the simplest part of the process. The syntax is:

```
PRINT#15,"M-E";CHR$(lo);CHR$(hi)
```

'Lo' and 'hi' specify the address to start executing at. The M-E command follows the same syntax rules as M-W. Use CHR\$() not ASCII; use semicolons, not commas; and DON'T PUT A COLON AFTER THE M-E.

This brings us to the BLOCK-EXECUTE (B-E) command, which is a combination of U1 and M-E. B-E loads a block from disk and executes it automatically. B-E requires a # channel like U1 and follows the same syntax too. Since B-E always starts executing at the beginning of the buffer, you don't have to specify an execution address. With M-E you can start anywhere in the buffer you wish. This is the only difference between using B-E and using U1/M-E.

There are two other alternatives to M-E: the job queue commands JUMP (\$D0) and EXECUTE (\$E0). Job queue commands are activated by putting their hex codes directly into the job queue located at \$0000-0005 in the 1541 (using M-W, for instance). This area is examined by the 1541's processor at every IRQ interrupt. If a valid job code is found, that task will be performed. Each job in the queue has a corresponding buffer and track/sector pointer. Job 0 (location \$0000) uses buffer 0 (\$0300-03FF) and gets its T/S info from location \$0006-7. Job 1 uses buffer 1 and location \$0008-9, and so on (job 5 has no buffer). Routines executed through the job queue should end by storing a status code back over their own job code in the queue and then jumping to the DOS's idle loop (wait for something to do). If the job code isn't wiped out, the routine will be re-executed over and over.

The JUMP (\$D0) command simply jumps to the beginning of the corresponding buffer and starts executing. The EXECUTE (\$E0) command is a deluxe command which positions the read/write head to the track specified in its T/S pointer, selects the proper density and waits for the drive motor to come up to speed before executing the routine in its buffer. This makes EXECUTE very convenient for program protection routines which check the disk for some type of custom

formatting. If you use M-E or JUMP, you must handle all this yourself. Of course, many times you'll want to change the density, move to a half-track, etc. anyway. What you do is up to you; the M-E, B-E, JUMP and EXECUTE commands are your keys to controlling the disk drive.

FAST LOADERS

Everyone knows how slow the 1541 drives are when it comes to loading programs. Actually, the drive can read data very quickly - a whole track only takes 1/5 second. The real bottleneck is the rate at which data can be transferred to the computer using serial communications. Serial means that the data is sent one bit at a time, as opposed to parallel communication, in which 8 bits are sent simultaneously. The serial bus cable contains six lines (wires). Only three of the lines are used for communication: ATN (attention), CLK (clock) and DATA. CLK should not be confused with the computer's 1 MHz timing clock. The other three lines are not used directly for communication: RESET, GND and SRQIN.

The ATN, CLK and DATA lines are connected to special chips in the computer and drive. These chips are accessed by reading or writing to particular memory locations. In the computer, port A of CIA chip #2 is used. This is located at \$DD00 (56576 decimal). In the 1541, port B of VIA #1 is used. This is located at \$1800 (6144). The serial lines appear as particular bits of these locations. The DATA and CLK lines have two bits each, one for input and one for output. The ATN line is a special one-way line that can only be controlled by the computer. The computer has an ATN OUT bit only and the drive has an ATN IN bit only. Another bit in the drive called ATN ACK (acknowledge) is used to restore the drive's circuitry and does not connect to the computer.

In the normal serial routines, ATN is used by the computer to signal the start of communication, regardless of whether it is going to send information (TALK) or receive information (LISTEN). During the actual data transfer, only the CLK and DATA lines are used. The TALKER sets the DATA line HIGH or LOW (+5 volts or ground) depending on whether the bit to be sent is a "1" or a "0". Then it sets the CLK line HIGH as a signal that the data is ready. When the LISTENER sees the CLK line go HIGH, it reads the data bit from the DATA line. This sounds pretty simple, and it is. However, we've ignored the "handshaking" procedure used to make sure the drive is present and tell it to TALK or LISTEN, etc. Handshaking adds time and complexity to the serial transfer routines.

Now let's see what can be done to speed up the transfer of a byte. The key idea used by all fast loaders is to send more than one bit at a time (you might call it "semi-parallel"). How many bits can we send at once? Well, since the ATN line is one-way from the computer to the drive, it's not suitable for general use. There are other problems with using ATN too. That leaves us with only CLK and DATA. DATA is already used to transfer data bits, but CLK is normally used to signal that the data bits are ready. If we use CLK to send a data bit too, how will the LISTENER know when the data bits are ready?

The answer is precise TIMING. The idea is to first get the drive and computer "in sync" at the beginning of the byte and then make sure that each takes the same amount of time to read or write a pair of bits via CLK and DATA. Instead of the LISTENER having to be told specifically when the bits are ready, it will depend on the TALKER to have them ready at the time it gets around to reading them. For this idea to work, several factors must be carefully planned. The most important is to get the two devices synchronized before the data transfer starts. For this we can use a simplified form of handshaking. In order to understand how it works, you first need to know a little more about the serial lines.

Suppose one device on the serial bus is trying to set a line LOW, but another one is trying to set it HIGH. Who wins? The answer is that if ANY device is trying to set a line LOW, it will go LOW. For a line to be HIGH, ALL devices on the serial bus must agree to "let" it be HIGH (by not setting it LOW). "Setting" a line HIGH is really a misleading term since no device can force a line HIGH by itself; all it can do by itself is force a line LOW. When we talk of "pulling" a line down, we'll mean setting it LOW. "Releasing" a line will mean letting it go HIGH if possible. Think of the emergency brake cord on a train. If any person pulls the line down, the train will SLOW down. For the train to stay at HIGH speed, everyone must refrain from pulling the line down!

One way to set up a handshake to synchronize two devices is by letting one device control the DATA line and the other control the CLK line. We can start by having the LISTENER pull the CLK line LOW and the TALKER pull the DATA line LOW. Both devices must also release the other's line so the other will have total control of it. Now the LISTENER waits while the TALKER gets a data byte ready. Once the TALKER is ready, it releases the DATA line HIGH, meaning "ready to send". Then it waits for the LISTENER to respond. When the LISTENER sees the DATA line go HIGH, it releases the CLK line HIGH, meaning "ready to receive". Now the two devices are synchronized and the transfer of data bits can begin. Note that both lines have just been released, so the TALKER can control them.

The TALKER must now break the data byte into 4 pairs of bits and send each pair using CLK and DATA. It will take some time for the TALKER to get the first pair ready, so the LISTENER has to delay approximately the same amount of time before it begins reading. This is another factor that must be planned carefully. As the LISTENER receives pairs of bits it must build them back into a byte. There are many ways to break down the byte and rebuild it, usually involving the shift (ASL, LSR) and rotate (ROL, ROR) instructions. The particular methods used will vary from one fast loader to another and are the major factors in determining the speed of the transfer. It is very important that the time required on each side be the same or very close. This may require that one routine be "padded" with some NOP's to slow it down. After the TALKER has sent the last pair of bits, it must delay a little to make sure the LISTENER has time to receive them. Then both devices must pull their respective lines LOW again to set things up for the handshake of the next byte. When all data has been transferred, they must release their handshake lines so normal serial communication is possible.

During data transfer, all interrupts on both sides must be disabled. IRQ interrupts can be disabled with SEI. The computer has a couple of very subtle interrupts that have to be disabled too. As the VIC chip fetches character data for the screen display, it "suspends" the processor every eighth raster line (see pp. 449 in the Programmers Reference Guide). This can be prevented by blanking the screen, which is common. Another method is to include a check in the byte handshake to wait until the VIC is not on the first or second raster line of each set of eight, by checking the VIC raster register at \$D012 (53266). This method leaves the screen visible. In addition to either method, sprites MUST be disabled through the register at \$D015 (53269).

To make a complete fast loader, you'll also need other routines to find the filename in the directory, move the head to the right track, set the density, find the right sector header and read in the data block. These routines are not simple, but readers of Program Protection Manuals Vol. I & II should be familiar with the concepts involved.

C64 KERNAL ROM REVISIONS

The Commodore 64 contains two 8K ROMs which hold the operating system routines. These two ROMs are commonly called BASIC and the KERNAL, although the BASIC routines 'spill over' into part of the KERNAL ROM. The BASIC ROM is located at \$A000-BFFF (40960-49151) in memory, and the KERNAL is located at \$E000-FF (57344-65535).

The C64 has undergone several revisions since it was first introduced. There have been some hardware modifications, but the most important changes have been in the operating system. Actually, the BASIC ROM is identical in all versions of the C64, including the SX64 portable model, so we're really talking about changes in the KERNAL. There have been 3 different versions of the KERNAL in the C64, plus a fourth version in the SX64. This month we're going to look at the different revisions Commodore has made to the KERNAL. Next month we'll show you how to make some revisions of your own.

One way to tell which ROM you have is to look on the ROM itself, which is a chip labelled 901227-XX, where XX is the revision number (this applies to the C64 only; you should be able to tell if you have an SX64!). An easier way is to type the following line: PRINT PEEK(65408). This location in the KERNAL (\$FF80 hex) is not used by any routines, but Commodore has put a different value there in each version. Another less commonly known location that serves the same function is 58540 (\$E4AC). These are the ONLY two bytes that are different in all four KERNAL versions. The values found there are given below (only Commodore knows why these particular values were used!).

REVISION	58540	65408
1	43	170
2	92	0
3	129	3
SX	179	67

Let's start with the differences between ROMs 1 & 2. More changes were made in ROM 2 than in any other ROM. The table below lists the areas which are different (all locations are given in hex) and a brief explanation of the effect of each change.

<u>LOCATION</u>	<u>CHANGE MADE TO ROM 2</u>
E119-1A	JSR to special BASIC CHKOUT routine (see E4AD)
E4AC	Unused location; see table above
E4AD-B6	BASIC CHKOUT; avoids problems PRINTing to nonexistent device
E4DA-DF	Fill color RAM with background color when clearing screen
E4E0-EB	Routine to wait only 8.5 secs for C= key on tape load/verify
E4EC-FF	PAL (International) RS-232 baud rate timer constants
EA0B-0E	JSR to \$E4DA patch above instead of using color white
ECCA-B	Set VIC raster interrupt to line 622 for PAL/NTSC check
ECD2	Clear VIC interrupts for PAL/NTSC check
F428-4C	Check PAL/NTSC flag and use proper RS-232 baud rate table
F459	JSR instead of JMP to RS-232 DATA-SEND-READY error routine
F762-66	Use \$E4E0 tape wait routine above instead of waiting forever
FCFC-FD	JSR to new CINT routine at \$FF5B (see below)
FDDD-F8	Set CIA #1 timer A (IRQ timer) based on PAL/NTSC flag
FEC2-D4	NTSC (North American) RS-232 baud rate timer constants
FF08-42	Modified RS-232 timing routines
FF5B-7F	Check VIC raster interrupt and set PAL/NTSC flag accordingly
FF80	Unused location; see table above
FF81	CINT (screen init) vector changed to \$FF5B routine above

Most of the changes in ROM 2 are related to the PAL/NTSC check, which allows the C64 to detect whether it is on an NTSC (North American) or PAL (International) system and adjust the IRQ and RS-232 timing accordingly. One other change is significant - ROM 2 fills color RAM with the background color rather than white when clearing the screen. This caused incompatibility problems with software, notably Wordpro, which poked characters directly onto the screen without setting their color. Other changes include creating a new BASIC CHKOUT routine to handle PRINTing to nonexistent devices correctly, and altering the tape load/verify routine so it only waits 8.5 secs for the C= key to be pressed after finding a file.

ROM 3 includes all of the features of ROM 2 except that the screen clear routine was modified again. This time color RAM is filled with the character color in effect when the screen is cleared. Commodore also fixed the infamous screen editor bug: with ROMs 1 & 2, if you go to the bottom of the screen, type 80 characters and then delete the last one, the computer will freeze up! A bug in the INPUT routine was also fixed - when the INPUT prompt was longer than one line, the prompt was taken as part of the input. A minor RS-232 parity bug was also fixed. The areas changed are given below:

<u>LOCATION</u>	<u>CHANGE MADE TO ROM 3</u>
E4AC	Unused location; see first table
E4D3-D9	Reset RS-232 parity when start bit detected
E4DB-DC	Fill color RAM with character color when clearing screen

E57C-90	Screen editor bug patch - JMP to \$EA24 to set color RAM addr
E591-99	INPUT bug patch - handle long prompts correctly
E622-23	JSR to \$E591 INPUT routine patch above
EA07-12	Modified routine to clear screen line
EF94-96	JMP to \$E4D3 RS-232 patch routine above
FF80	Unused location; see first table

The SX64 ROM contains all the features of ROM 3. The major change is that any attempt to use tape results in an 'ILLEGAL DEVICE' error since the SX has no cassette port. The default colors and start-up screen were also changed, and the SHIFT-RUN/STOP key was programmed to perform a LOAD":*",8 instead of just LOAD. The areas changed are:

<u>LOCATION</u>	<u>CHANGE MADE TO SX64 ROM</u>
E479-93	Start-up screen changed to "SX-64 BASIC V2..."
E4AC	Unused location; see first table
E535	Default character color changed to blue (code 06)
E5EF	No. characters in SHIFT-RUN/STOP command changed to 15
E5F4-F5	Location of SHIFT-RUN/STOP command changed to \$F0D8
ECD9-DA	Default border color cyan (03); background color white (01)
F0D8-E6	SHIFT-RUN/STOP message changed to LOAD":*",8 (CR) RUN (CR)
F387	Patch OPEN routine to give "ILLEGAL DEVICE" error for tape
F4B7	Same as above, for LOAD/VERIFY routine
F5F9	Same as above, for SAVE routine
FF80	Unused location; see first table

This concludes our look at the C64 ROM versions.

MODIFICATION OF THE DOS (General)

WHERE TO START: First, it will be necessary to have access to an EPROM programmer (burner). This is a device that will allow you to burn your own routines into an EPROM (AN IC CHIP THAT MAY BE PROGRAMMED). These EPROMs may be substituted for the chips in the disk drive.

The chips (in the disk drive) that you can modify are located near the power regulator (the large heat sink at the right rear of the drive). Each chip is 8K and contains half of the operating system. The chip marked 325302 contains the code from \$C000 to \$DFFF. This chip may or may not be socketed (removable) on some disk drives. Fortunately, most of this code does not ever need to be modified. The chip marked 901229 contains the code from \$E000 to \$FFFF. This chip has always been socketed in every disk drive that I've seen.

The EPROM programmer that we use here is the PROMENADE by Jason-Ramheim. All commands given here will be for the PROMENADE.

1) Follow the instructions included with the PROMENADE to zero (Z) the programmer.

- 2) Remove the 901229 CHIP from the disk drive. Insert the CHIP into the PROMENADE. Use the following command to read the CHIP.
8192,16383,0,48 (return)
- 3) The code from the CHIP will be stored in the computers memory from \$2000-\$3FFF.
- 4) You may now use your ML monitor to examine and modify the DOS.
- 5) The very first place to modify the DOS is at \$EAE4, \$EAE5, \$EAE8 and \$EAE9.

Change these locations to \$EA (NOP). This is the area in memory that test the ROM, it performs a checksum test on all 16K of the DOS. If you modify the DOS it will be necessary the temporarily remove the ROM test. Otherwise, the drive will not pass the checksum test.

- 6) You may now freely modify the DOS and change all code that you wish. The most important place to start modifying the code is in the interrupt routine for the disk drive (\$FE67). This routine is where the actual work gets done. Most of the rest of the DOS is for house-keeping chores. Use the ABACUS book and examine the interrupt routines. Remember, once that you down load the DOS into the computer the code will reside from \$2000-\$3FFF in the computers memory. So, if you want to change \$EAE4 in the DOS, you will have to change \$2AE4 in the computer.
- 7) To save the code back out to an EPROM (MCM68764) use the following command:
8192,16383,0,48,15

To save the code out to a 2764 EPROM use the following command:
8192,16383,0,5,7

The 2764 is a low cost EPROM (\$10.00 or less). The MCM68764 is an expensive (\$29.00 or more) EPROM. The MCM68764 is a pin for pin replacement for the CHIP that is used in the disk drive. The 2764 requires the use of an adapter to function properly in the disk drive (part #AD \$8.95 from CSM). As you can see, it will be very cost effective to buy a couple of adapters. Then use the low cost 2764 EPROMs.

MODIFICATION OF THE DOS (40 Tracks)

HARDWARE MODIFICATION OF THE OPERATING SYSTEM FOR MORE THAN THIRTY-FIVE TRACKS OR EXTRA SECTORS.

Caution: Some drives may not be physically able to go to track 40. The read/write head may become stuck at track 38 on some drives. This is not serious, just go into the drive and free the head with your hand if it gets stuck.

This method will require the 'burning' of replacement EPROMS for the disk drive. You will be able to add extra tracks or vary the number of sectors on a track when you use this technique. One problem will be evident when you change the number of tracks on the disk: You will only be able to list the directories of disk with same number tracks as the drive is set up for. For instance: if you have a forty track drive you can only list the directories of the forty track disks. This is due to the way the 1541 wants to find the BAM, NAME, and ID of the disk. As you add extra tracks you must also add room for increased area in the BAM (each track requires 4 bytes for the BAM). The BAM will be expanded into the area normally used by the NAME and ID of the disk. The NAME and ID must be located immediately following the BAM of the last track. If you add five tracks to the disk you must increase the BAM by 20 bytes. The NAME and ID will also be moved 20 bytes on the disk (the drive will automatically locate the NAME and ID immediately after the BAM).

The items to change for the extra tracks will be the location of the end of the BAM (four bytes for each track) and the comparisons for the maximum number of tracks. The other area of memory to change when ever you modify the DOS is the ROM TEST (\$EAE4-\$EAE9). This is where DOS checks the ROM to insure that there has not been any malfunctions of the operating system. We will totally bypass the ROM test, this allows you make any modification that you wish. These are the following locations to change for a forty track drive.

\$D08C CHANGE TO	\$A4	(LOCATION OF BAM)
\$EAE4 CHANGE TO	\$EA	(ELIMINATE ROM TEST)
\$EAE5 CHANGE TO	\$EA	(ELIMINATE ROM TEST)
\$EAE8 CHANGE TO	\$EA	(ELIMINATE ROM TEST)
\$EAE9 CHANGE TO	\$EA	(ELIMINATE ROM TEST)
\$EEEE CHANGE TO	\$A4	(LOCATION OF BAM)
\$FD90 CHANGE TO	\$29	(MAX. # OF TRACKS +1)
\$FE88 CHANGE TO	\$A4	(LOCATION OF BAM)
\$FED7 CHANGE TO	\$29	(MAX. # OF TRACKS +1)

If you wish to modify the number of sectors per track change the code from \$FED1-\$FED4. Each byte represents the number of sectors on the different tracks (\$FED1 is for TR 31-35; \$FED2 is for TR 25-30; ETC.). There are certain limits as to the number of sectors on a disk. Don't try to add to many, it won't work.

Bytes \$FED8-\$FEDA indicate where the sector change will occur (tracks 31, 25 & 18).

How to make the modifications to the disk drive:

First locate and remove the ROM chips from the disk drive. The \$C000-\$DFFF chip is marked 325302, the \$E000-\$FFFF chip is marked 901229. Both chips are located at the rear of the circuit board. The \$C000-\$DFFF chip is not socketed on some disk drives. If your chip is not socketed, don't try to remove it from the board, just modify the \$E000-\$FFFF chip. The drive will not format disks nor will the BAM operate properly if you don't modify the both chips. But, you will at

least be able to read and write 40 tracks if you can only modify one of the chips.

Insert the \$C000-\$DFFF ROM into your PROMENADE. Down load the chip memory into the computer. Use a ML monitor to modify the code as specified above. Then remove the original chip and insert an erased EPROM into the PROMENADE and burn the EPROM. Insert the EPROM into the proper socket, then use the same procedure on the other chip.

The total time required to complete the process is less than 1 hour and can provide some very interesting results. Don't be afraid to experiment with computer or disk drive. Try some modifications on your equipment and see what fun you can have.

SAVE @ BUG

Dr. Phil Slaymaker, the author of PEEK-A-BYTE V2.0, has proven the existence of the infamous SAVE @ (save with replace) bug that has been rumored for years on Commodore disk drives. In two recent articles in COMPUTE!, he explains why it happens and how to avoid it as much as possible. He also gives a sample program which will trigger the bug.

There doesn't seem to be a foolproof method of preventing the bug, just making it very unlikely. First, you should use the drive number (i.e. 0:) in ALL disk commands, not just SAVE @. Second, make sure all files in the drive are closed before going on to other disk operations. Leaving files open in the drive, even if they're closed in the computer (via CLR or SYS 64738 for instance) can cause the bug too. If you still want to use SAVE @ after all this, you should RESET the drive or issue it a "UI" or "UJ" command before the SAVE @. If you're ready to abandon SAVE @ entirely because of the risk, consider this. Dr. Slaymaker says that the bug is not confined to SAVE @ exclusively - it can happen with other disk commands, even a separate scratch and SAVE sequence!

Our research has shown that the new 1571 drives have the same bug, in both 1541 and 1571 mode (at least with version 3 of the ROM). Several products claim to fix the SAVE @ bug, including FLASH! and STARDOS, but this doesn't appear to be true. Commodore themselves could have reduced the chances of the bug occurring by adding more RAM to its drives. There are indications that they considered this on the 1571 drives - the manual claims in one spot that the drive has 4K of RAM, whereas physical inspection of the drive's innards turns up

CSM Software Inc. is proud to present the ULTRA LIGHTNING LOADER. This fantastic utility speeds up your disk drive at least 500%. Unlike similar utilities on the market, ULTRA LIGHTNING is totally compatible with all current Commodore 64 software INCLUDING COPY PROGRAMS AND QUICK-LOAD UTILITIES! This means your nibble or fast copy programs will now be 5 times faster. If you already have a

quick-load type utility ULTRA LIGHTNING will speed it up even more! This can give you an increase in speed of up to 25 times the normal 1541 speed. ULTRA LIGHTNING is a special treat for our newsletter subscribers, at no charge. I'm sure you'll agree it is worth many times its cost. To put ULTRA LIGHTNING to use on your computer, follow this procedure:

1. Type in the program listed below. Be especially careful to type the data statements correctly.
2. When you are finished typing the program, DO NOT try to run it. Save the program to a disk first.
3. Turn off the computer and disk drive.
4. Turn the drive back on first, then the computer.
5. Load in the ULTRA LIGHTNING program and run it. The program will check the data statements and tell you if you have made any errors.
6. If there are no errors, the program will ask you for your drive's device number.
7. The program will then download a special routine into the disk drive, and also latch itself into the computer's operating system.
8. When it is finished, you are ready to load your copy utility or other program. Don't attempt to examine the code in memory or execute any statements other than a LOAD statement.
9. That's all there is to it.

```

1 CK=0 :FOR I=1 TO 134 :READ A :CK=CK+A :NEXT :RESTORE
2 IF CK<>19048 THEN PRINT "ERROR IN DATA STATEMENTS" :END
10 PRINT CHR$(147) TAB(9) "ULTRA LIGHTNING LOADER" :PRINT :PRINT
20 INPUT "ENTER DISK DEVICE NUMBER"; DN :IF DN<8 OR DN>11 THEN 20
30 PRINT :PRINT :PRINT "INITIALIZING - PLEASE WAIT ..." :PRINT
40 OPEN 15,DN,15,"IO"
50 OPEN 2,DN,2,"#"
60 PRINT#15, "M-W" CHR$(29) CHR$(0) CHR$(1) CHR$(DN)
70 PRINT#15, "M-W" CHR$(35) CHR$(0) CHR$(1) CHR$(73)
80 PRINT#15, "M-W" CHR$(0) CHR$(4) CHR$(1) CHR$(99)
90 FOR I=1 TO 32 :READ A :PRINT#2,CHR$(A); :NEXT
100 CLOSE 2 :CLOSE 15
110 FOR I=0 TO 101 :READ A :POKE 52992+I,A :NEXT
120 SYS 52992
130 PRINT "ALL DONE. NOW LOAD YOUR PROGRAM" :NEW
140 DATA 16,129,43,23,193,34,240,54,117,30,44,152,99,32,202,15
150 DATA 63,219,55,19,233,59,172,70,199,41,71,210,13,55,151,47
160 DATA 169,15,162,255,143,48,3,169,207,162,255,143,49,3,96,169
170 DATA 4,141,32,208,141,33,208,160,63,191,38,207,73,170,32,202
180 DATA 241,136,208,245,240,254,139,139,235,226,233,254,229,237
190 DATA 53,138,138,138,138,138,138,138,138,138,138,138,138,138
200 DATA 138,138,138,187,187,167,249,141,230,229,229,236,138,230
210 DATA 227,248,250,235,52,138,138,138,138,138,138,138,138,138
220 DATA 138,138,138,138,138,187,187,187,187,187,187,187,187,187,57

```

UNIVERSAL BACKUP PROGRAM

This month we are pleased to present the UNIVERSAL BACKUP PROGRAM from Lirpa Loof Software. Another CSM exclusive! Those of you who remember the incredible ULTRA LIGHTNING program from last year will be equally impressed with this brand new utility. The Universal Backup Program allows you to obtain an achival backup of ABSOLUTELY ANY PROGRAM currently available commercially - NO EXCEPTIONS. You're probably saying, "I've heard that one before". Well, this time it's TRUE!

Using the Universal Backup Program is simple. Just type in the boot program below and save it to a disk. When you RUN the program, it will request the name of the program which you wish to obtain a backup for. Next it will request the name of the software company that produces the program (in case of duplicate program names). You should have your printer connected and on-line for best results. Otherwise, you may have to perform certain steps in the backup procedure manually.

The Universal Backup Program works with all Commodore-compatible disk drives and all properly interfaced printers. IMPORTANT NOTICE: The Universal Backup Program is for archival use ONLY! The makers of the Universal Backup Program do not condone software piracy!

```
1 FOR I=1 TO 3 :READX$ :A$=A$+X$ :NEXT :RESTORE
2 FOR I=1 TO LEN(A$) :A=A+ASC(MID$(A$,I)) :NEXT
3 IF A<>5488 THEN PRINT "ERROR IN DATA" :END
10 INPUT "NAME OF PROGRAM";N$ :PRINT CHR$(147)
20 OPEN 15,8,15 :PRINT#15,"M-E"CHR$(4)CHR$(208)
30 OPEN 2,4 :PRINT#2," "
40 IF ST=-128 THEN CLOSE2 :OPEN 2,3 :GOTO 60
50 PRINT "MAKE SURE PRINTER IS ON-LINE"
60 FOR I=1 TO 4 :READX$ :GOSUB1000 :A$(I)=Z$ :NEXT
70 PRINT#2, CHR$(13)
80 PRINT#2, A$(1) CHR$(13)
90 PRINT#2, A$(2) N$ A$(3) CHR$(13)
100 PRINT#2, A$(4)
110 CLOSE2 :CLOSE15 :END
200 DATA ";>-<,6-,E"
210 DATA ":1<30,;>+6,<+<7:<4<90-<+7:</*-<7>,:<09<+7:<+
220 DATA "<+>-<76)>3<=><4*/<+6,4W,VQ<+7>14<+0*Q"
230 DATA ",61<-:-:3&S"
1000 Z$=""
1010 FOR J=1 TO LEN(X$)
1020 Z$=Z$+CHR$(127-ASC(MID$(X$,J)))
1030 NEXT
1040 RETURN
```

HOW TO GET THE MOST OUT YOUR SNAPSHOT UTILITY

This article gives some personal observations on how to best use SNAPSHOT 64, ISEPIC and CAPTURE (if it ever arrives). We know some readers will have other suggestions on how to get the most from their snapshot type of utility. By snapshot type of utility we mean any or all of these products.

First and foremost, pay attention to how the program loads in. If you can, watch the head move from track to track. Watch the screen and the border changes. Listen to the speaker on your TV or monitor. Make good written notes about where and when things happen. Don't forget to make written notes, especially if you're having trouble making the archival backup.

1. Watch the head move. Go ahead and take the cover off of your drive so that you can see the position of the head. If you know where the protection key is located on the disk, you will be able to make an educated guess as to when the program makes its protection check. If you know that the program checks its protection early on, use your snapshot to stop the program as soon as the program pauses or when the disk drive stops spinning. If the program loads in a chunk of the main program, then checks the protection and loads in another chunk of the main program, try to use your snapshot just before the second chunk of the program gets loaded in. This point can usually be found by watching the movement of the head.
2. Watch for changes in the screen or border of the monitor, while the program is loading. Many times the programmer will set up the program so the screen changes right after the protection check. Occasionally it will be no more than a flicker or a slight change of color. Other times it will be more dramatic, such as a title screen, a menu or hi-res graphics. Remember, programmers are human too - they like to know if a protection scheme is working properly. By watching the screen, many times you will find where the programmer gives himself away.
3. Listen for the sound chip to be initialized. This will normally be accompanied by a small 'POP' from the speaker. If your program is dependent upon sound (or even if it's not) for its proper functioning you may find that by stopping the program just before the sound chip is initialized you will be in the proper spot. Many times you will have to turn the volume way up on your monitor to hear the 'POP'.

If your program works properly, but no sound, try getting the program to go back to the menu. Many programs are designed to restart the menu by using the RUN/STOP and/or the RESTORE keys. Use your owners manual as a guide when finding the proper sequence to restart the program.

Occasionally there may be only a second or two where program may be stopped. Try the old 'trial and error' method of stopping the program. Make notes of how it functions and what results are obtained. Usually you can narrow down the proper point. Of course, trial and error can be time consuming, especially with ISEPIC.

Use the code inspection function of your snapshot utility to find where the program is executing at. Many times you will stop the program in the middle of a KERNAL routine. Then you will need to examine the STACK to find out what part of the program called the KERNAL routine (actually the stack pointer points to the next free byte and the addresses contained on the stack need to be incremented by one). Look in the general area of the program's execution location for the protection scheme.

WORKING 'INSIDE' THE DISK DRIVE

In order to work inside the disk drive, it will be necessary to understand how to use an 'essential' tool. The tool that we are speaking of is called 'DRVMON64' by STARPOINT SOFTWARE (the DISECTOR people). DRVMON64 is probably the most powerful tool available for working inside the disk drive. DRVMON64 is also one of the most overlooked tools around. DRVMON64 is copyrighted by STARPOINT SOFTWARE and contained on the DISECTOR disk. One interesting fact about this program is that it is not copy protected.

Two other items that you should have in your 'tool kit' are the books INSIDE COMMODORE DOS by Richard Immers and Gerald Nuefeld and THE ANATOMY OF THE 1541 DISK DRIVE by Abacus. With these two books and DRVMON64 there is virtually nothing that cannot be accomplished inside the 1541. Inside Commodore DOS is an excellent book, well written, easy to understand and by far the most comprehensive book on the disk drive.

HOW TO USE DRVMON64

DRVMON64 is a ML monitor that will work equally well in the disk drive or the computer. It is possible to transfer data between the disk drive and the computer. It is possible to assemble or disassemble code in the computer or directly in the drive's memory. As a matter of fact, all of the ML monitor functions that are available in the computer are available in the drive with DRVMON64.

If you don't currently have a copy of the DRVMON64 we would strongly suggest that you obtain one. Since it is a copyrighted program we can not supply you with a copy of it. If you already own DRVMON64 we suggest that you get out your copy of the DISECTOR manual and read the DRVMON64 instructions before proceeding any further.

Load and execute the version of DRVMON64 that resides at 49152 (\$C000). This is the version of the program that we will use for our discussion. When DRVMON64 is first executed the monitor will be working in the computer's memory. To use the monitor in the disk drive simply type 08 (0 the letter, not 0 the number) then RETURN. You will notice that there is a ']' (bracket sign) next to the cursor, this indicates that you are in the drives memory. Let's try some real easy commands. Type 'M 0000 003F' (RETURN). The code that you see is the actual code from the disk drives memory starting at \$0000 and ending at \$003F. You may also

use the other commands (A, D, F, G, etc.) when you are in the disk drive. Try the following command 'D FCAA' (RETURN), you will now see a portion of the ROM memory disassembled. DRVMON64 is fast, it is easy to use and it is probably the best tool for examining the disk drives memory.

For now we will concentrate on one area of the drives memory, the job command queue #0 located at \$0000 and the track and sector for buffer #0 located at \$0006 (track) and \$0007 (sector). The disk drive will use these locations to perform many of its tasks. When the disk drive reads a block of data from the disk it will use the job command queue located at \$0000 to perform this function. This location is checked by the disk drive through its interrupt routine (IRQ). If the proper value (command code) is stored at location \$0000 the drive will execute the function that corresponds the command code. Following is a table that describes the command codes and their meanings. All values are in hex:

80	READ A SECTOR
90	WRITE A SECTOR
A0	VERIFY A SECTOR
B0	SEEK A TRACK (ANY SECTOR)
B8	SEEK A TRACK & SECTOR
C0	BUMP - FIND TRACK 1
D0	JUMP TO ML PROGRAM IN BUFFER
E0	EXECUTE CODE IN BUFFER - FIRST THE DRIVE WILL BRING THE DISK UP TO SPEED AND SEEK THE PROPER TRACK, THEN THE CODE IN THE BUFFER WILL BE EXECUTED.

The track and sector number should be stored at locations \$0006 and \$0007 respectively prior to storing the command code at location \$0000. When we use DRVMON64 it is possible to set all the values at the same time and let the program do the rest for us. So much for theory, let's have some fun! Format a disk and leave it in the disk drive. This way we can have something to work with. Do not experiment with a valuable disk. We will use the M command of DRVMON64 to perform the following experiments. For the sake of clarity it may help to remove the cover of the disk drive and the metal shield. This will enable you to see the actual results (of the R/W head) that we are going to obtain. Type in the following lines directly next to the] (bracket sign).

]@I (RETURN)

This will initialize the disk drive. DRVMON64 also contains a built in DOS wedge. Now type in the following line. Don't add any extra spaces or data to this line.

]F 0300 03FF 00 (RETURN)

This command will fill data buffer 0 (located from \$0300 to \$03FF) with 00's. This gives us a fresh work space. Next type in the following line.

]0000 80 00 00 00 00 00 01 05 (RETURN)

Let's examine the above line. The \$80 (at location \$0000) specifies read a block. The \$01 (at location \$0006) specifies the track number. The \$05 (at location \$0007) specifies the sector number. If everything went as expected the drive will come to life, the disk will start spinning, the head will move and track 1, sector 05 will be read into buffer 0 (located from \$0300 to \$03FF). Before we examine the code from \$0300 to \$03FF let's be sure that the drive was able to properly read the data. To do this it will be necessary to examine the job command queue for an error message. After a command is processed by the drive, the DOS will store an error code back in the command code queue. The error codes are interpreted as follows:

01	NO ERROR - JOB COMPLETED
02	HEADER BLOCK NOT FOUND
03	SYNC NOT FOUND
04	DATA BLOCK NOT FOUND
05	DATA BLOCK CHECKSUM ERROR
07	VERIFY ERROR (AFTER WRITE)
08	WRITE PROTECT ERROR DURING WRITE
09	HEADER BLOCK CHECKSUM ERROR
0A	DATA BLOCK TOO LONG
0B	ID MISMATCH ERROR
10	BYTE DECODING ERROR

Type 'M 0000 0007' (RETURN). We will now be able to examine the contents of location \$0000 for the error message. If there was no error the value of \$01 will be contained at location \$0000. If any other value is contained there, repeat the procedure starting with the '@I' command. Use the 'M' command to examine memory from \$0300 to \$03FF. This is the area of memory that contains the data block that was read.

Now that you can read a normal data block into memory let's go a little farther. Just by substituting a \$90 at location \$0000 we can write the data at \$0300-\$03FF out to any sector. Try using the other commands to Verify, Seek or Bump. On word of caution though, if you use the JUMP or EXECUTE commands be sure that there is a valid ML routine at location \$0300. If you don't have a valid ML routine located there the drive will lock up. A power-off or a RESET will be required to restore your drive to normal.

It is not necessary to use a valid track or sector number at memory locations \$0006 & \$0007. Try repeating the above procedure with a track number of \$24 (decimal 36). It is possible to move the head to any track on the disk (1 to 40) and read data from the disk. The data that will be read must have been written in the standard 1541 format. Otherwise an error will be returned and data read will not be valid. It is interesting to note that the head will not 'BUMP' when errors are encountered when using the job command queue to execute direct commands. Try removing your disk and execute the read command. The head will not BUMP when an error is encountered if we are directly executing the command.

If you wish to use the EXECUTE command it will be necessary to write your own ML read and write routines. With the use of these routine it is possible to

read and write data to the disk ANYWHERE from track 1/2 to track 40 1/2. The routines necessary to perform these operations will be discussed in detail in the PROGRAM PROTECTION MANUAL VOLUME II.

One other very interesting area in the disk drive's memory is the disk controller port B, location \$1C00. Memory location \$1C00 is where we can control the stepper motor, the drive motor and the density selection. Each bit of the byte at location \$1C00 has a special function. Following is a table that illustrates the function of each bit

BIT	VALUE	FUNCTION
0	\$01	BITS 0 & 1 ARE CYCLED TO STEP
1	\$02	(MOVE) THE HEAD IN AND OUT
2	\$04	MOTOR ON AND OFF (1 BIT IS ON)
3	\$08	DRIVE BUSY LED (LIGHT)
4	\$10	WRITE PROTECT DETECT
5	\$20	DENSITY SELECT
6	\$40	DENSITY SELECT
7	\$80	SYNC DETECT

Let's just try another little experiment. We will be changing the values contained at location \$1C00. Be sure that the cover is removed from your drive so that you may see the half tracks. Initialize your drive (@I) then type in the following line from DRVMON64. Be sure you are working in the drive (}).

```
JM 1C00 1C07 (RETURN)
```

You will see that the value contained a location \$1C00 is a \$D2 (binary %1101 0010). Examining the binary value we can see that bit 2 contains a 0. This means that the drive motor is off (remember bits are numbered from 0-7). If we want to turn on the drive motor all we have to do is set bit 2 to the value of 1. This will result in the binary value of % 1101 0110 (hex \$D6). To store the value of \$D6 in memory location \$1C00 use the following command.

```
J:1C00 D6 (RETURN)
```

The drive motor will come on and disk will spin. The disk will continue to spin as long as bit number 2 of \$1C00 is set to a value of 1 (HEX \$D6 = BINARY % 1101 0110). To turn the motor off move the cursor up to the D6 and change it to the value of D2 (then press RETURN). The motor will turn off. In order to move the head in half tracks it will be necessary to cycle the 0 & 1 bits in the following fashion. First initialize the drive (@I). Then use the following command:

```
J:1C00 D6 (RETURN)
```

Move the cursor back up to the D6 and change the value to D7, then D4, then D5, then D6, then D7, then D4, then D5, then D6, etc. To move the head in half tracks all we have to do is cycle bits 0 & 1 of location \$1C00. Use the following sequence of bit values 00, 01, 10 ,11, 00, 01, 10, 11, etc. Start with the

appropriate bit pattern (10) and cycle through the others. Change only those bits that directly affect the location of the head (bits 0 & 1). The first D6 will just turn the drive motor on, the D7 will move the head 1/2 track in, the D4 will again move the head 1/2 track in, etc. If you were to analyze the bit pattern used at location \$1C00 it will become apparent that all we are doing is cycling bits 0 & 1 to move the head. Conversely, if we wish to move the head out from track 18, we would do the following. Initialize the drive (@I) so that we start with the head at track 18. Type in the following:

```
J:1C00 D6      (RETURN)
```

This will turn on the drive. Then move the cursor back up to the D6 and change the value to D5, then D4, then D7, then D6, then D5, then D4, then D7, then D6, etc. In order to move the head in the opposite direction all we have to do is cycle bits 0 & 1 in the reverse order.

The internal DOS of the disk drive will take care of cycling the bits to move the head during normal operation. The DOS will also perform the action faster and more accurately than we can with our crude little experiment here. We hope that you have learned a little about the inner workings of the disk drive. Don't be afraid to experiment; try moving the head to various tracks. If you go too far the drive may have to be initialized. Just don't perform too many 'BUMPS' and you should not experience any problems. Be sure to remove the cover of your drive so that you may see the Read/Write head move to these 'exotic' locations.

HOW DO YOU WRITE A TRACK ADJACENT TO A HALF-TRACK?

On the 1541 it is very difficult (if not impossible). This is due to the write with erase feature of the drive that erases a guard band between tracks. When you write a track, the adjacent half-track will be erased by the guard band. Unless.... you want to try modifying your drive. This is a procedure that sometimes works (sometimes it don't). If the track and the half-track contain identical information (as does most of ARTs disks) this method may help. NOTICE: ANY MODIFICATION TO THE DISK DRIVE MAY RESULT IN DAMAGE AND/OR INJURY. CSM SOFTWARE INC. WILL NOT ACCEPT ANY RESPONSIBILITY FOR DAMAGE AND/OR INJURY AS A RESULT OF THIS PROCEDURE OR ANY OTHER PROCEDURE!

1. Locate the connector to the read write head on your disk drive. This is a black plastic connector that uses wires 1, 2, 3, and 5 (4 is not used).
2. Remove the #3 wire (normally the yellow wire) from the connector. This wire supplies the current to the ERASE portion of the R/W head, thereby preventing the guard band from being erased.

If your R/W head produces a wide enough track it just may be interpreted by the drive as having the same data on the track and the half-track.

Originally these disks that contain data on the track and the adjacent half-tracks may have been created on drive with a 96 track per inch (tpi) R/W head assembly. The 1541 uses a 40 tpi R/W head assembly. As you can imagine, the drive that uses the 96 tpi R/W head will produce a much narrower track than one produced with a 40 tpi head. If the actual track is narrower the guard band will be narrower. With this information in mind you can see how it is possible to write adjacent tracks and half-tracks. With our little hardware modification it is possible to force the disk drive into writing a little track wider than normal. This extra wide track may be interpreted by the drive as data on the track and on the half-track. Just keep in mind that this procedure does not always work as outlined. This is probably due to variations of the drive and of the disk media.

3. Be sure to reinstall the wire after testing the procedure. AS WITH ALL HARDWARE MODIFICATIONS... USE CAUTION!!

WRITE PROTECT AND DEVICE NUMBER SWITCHES

This month we present two hardware projects for your 1541 disk drive: a device number switch and a write protect control switch. The device number switch lets you set your drive to device 8 or 9 as needed. The write protect control switch is a unique modification that lets you write protect a disk without using a tab. Not only that, you can also write onto a disk that DOES have a tab, if you want! A third setting returns your drive to normal write protect operation, i.e., controlled by the sensor.

WARNING: As usual when you modify your drive, you can damage it if you're not careful. Don't make any modifications unless you're reasonably sure of yourself. And of course, if you do decide to modify your drive, CSM can't take responsibility for any damage that might occur. With this understood, we can reassure you that these modifications are very simple and can be done easily in 20 minutes or so.

TOOLS & MATERIALS: You'll need a low-watt soldering iron (35 watts or less) and some resin-core solder (not acid-core!). You'll also need some insulated wire (24-gauge is plenty thick enough). Unless you're the macho type with strong teeth, you'll need a pair of wire cutters/strippers too. A small Phillips screwdriver is required to disassemble the case. Mounting the switches on your drive's case will require a drill and appropriately sized drill bit. Finally, you'll need one SPST switch for the device number (e.g. Radio Shack part no. 275-324 or 275-624). For the write protect control, you'll need one SPDT CENTER-OFF switch (275-325) and a pair of microclips (270-370).

GETTING STARTED: Disconnect all power and serial bus cables from the drive. Turn the drive upside down and unscrew the four screws that hold the top of the case on. Remove the top. Unscrew the two screws that hold the metal shield on and remove the shield. Reverse these steps to reassemble.

DEVICE NUMBER SWITCH: Using the appropriate diagram below, locate the pair of device number jumper pads on the circuit board. Some boards have other jumper pads, so be sure you have the correct pair. Each pad of the pair consists of two semicircular halves, normally connected in the middle by a thin line (jumper). If both jumpers are left intact, the drive will be device 8. Cutting the jumpers will change the device number. Cutting just jumper #1 will change the drive to device 9, cutting just #2 will make it device 10, and cutting both will make it device 11. We'll only need jumper #1 to select between device 8 and 9 (make sure jumper #2 is connected). On the older long boards, #1 is the closest of the pair to the rear of the board. On the newer short boards, #1 is closest to the front. Using a small jeweler's screwdriver or dull knife, carefully cut through jumper #1. Make sure the jumper is cut completely through down to the board. Don't use a lot of force or you may slip and cut other lines on the board!

Now pick a spot on the case to mount your device number switch. Cut two pieces of wire long enough to run from the spot you've selected to the jumper pad, with a very generous margin for error. Strip about 1/16" of insulation from the ends of each wire - just enough to make a connection without danger of shorting. Solder one wire to each of the connectors on the SPST switch. Now solder the other end of each wire to a different half of the jumper pad - it doesn't matter which wire goes on which half. When soldering directly on the board, be sure to work quickly but carefully so as not to damage nearby components.

To finish up the installation, drill a hole in the case at the spot you selected and mount the switch. Reconnect the power and serial bus cables, and try to load a program from device 8. If you get "device not present", the switch is set for device 9. You'll probably want to mark which position corresponds to which device number. To change the device number, flick the switch and do one of the following: turn the drive off and back on; reset it; or send it a "UI" command.

WRITE PROTECT CONTROL SWITCH: Start by locating the white 15-pin connector along the left edge of the board (it's by far the longest connector on the board). The fourth pin from the front end of the connector is pin 12 (the pins are numbered starting at the rear end). The wire connected to this pin is orange on all the drives we've seen. This wire comes from the write protect sensor. Unplug the connector from the metal pins on the circuit board. There is a set of narrow slots on the side of the connector, one slot for each pin. You can see a metal contact inside the slot. By pressing on the upper end of the contact with a very small screwdriver, you can pull the wire and contact neatly out of the connector. Alternatively, you can just cut the wire and strip the end from the sensor.

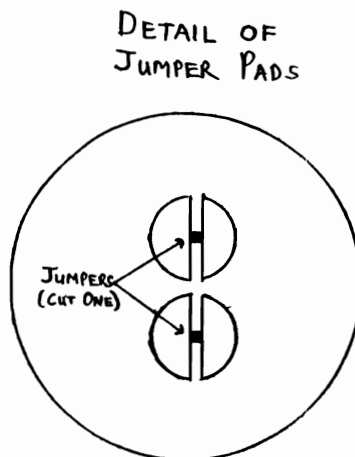
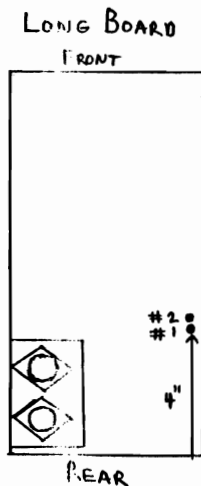
Now pick out a spot on the case for this switch. Cut three pieces of wire the appropriate length (plus safety margin) and strip the ends 1/16". Solder a wire to each of the three contacts on the SPDT/center-off switch. Solder a microclip on the remaining end of the center switch wire. Clip this microclip to pin 12 on the circuit board, at the base of the metal pin. Make sure the microclip's hook does not contact any other pins. Plug the connector back onto the pins. It should make firm contact despite the microclip. Now solder the

second microclip to the end of one of the other two switch wires - it doesn't matter which one. Clip this microclip to the wire from the sensor (the contact you pulled out of the connector earlier or the stripped wire end). Wrap this connection with electrical tape to insulate it. Finally, unscrew the ground screw at one of the corners of the circuit board. Wrap the end of the last remaining wire around the screw and screw it back in. Now drill a hole in the case and mount the switch.

The SPDT/center-off switch has three positions. With the switch set to one side the drive will operate normally, i.e., the write protect status will be controlled by whether the disk's notch is covered or not. Flipping the switch to the other side position will protect the disk from being written to even if its notch is uncovered. Setting the switch in the center position will allow you to write "through the tab" to any disk, even if the notch has never been cut! You'll definitely want to label which position is which.

There is one thing about using this switch that you must remember. Unless the switch is set to the "normal" position, the drive will not know when you change a disk. If the second disk has a different ID than the first, you'll get an Error 29, DISK ID MISMATCH, when you try to access the disk. This can be cured by flipping the switch from the side to the center position (or vice versa) after inserting the disk. The drive will sense that the write protect status has changed momentarily, which happens normally any time you change a disk. Alternatively, you can send the drive a "UI" or "IO" command. If the two disks have the same ID, it is possible in rare circumstances to write the wrong BAM onto the disk if you don't take these precautions. The best idea is to set the switch to the normal position before inserting or removing a disk.

Location of Device Number Jumpers:



PROGRAM BACKUP PROCEDURES

The following was submitted by a subscriber from Lynwood, CA.:

The program is ACROJET, (c) 1985 by Microprose, Inc. and William F. Derman, Jr.

TYPE OF PROTECTION: Track 35 is heavily protected with nonstandard DOS data.

HOW TO COPY: This disk is could not be duplicated by any of the nibblers we tried. Some of the new parameter copiers such as Fast Hack'em 3.0 can make a working copy of the program.

HOW TO MAKE A WORKING COPY WITH NO ERRORS ON THE DISK:

- 1) Copy the disk using a copier that does not put errors on the disk.
- 2) Load and run a T/S editor.
- 3) Read in track 17, sector 1 from the COPY disk. Change the value at relative byte \$32 (50 decimal) from \$F0 to \$D0 . Write the sector back to the COPY.
- 4) You're done - Happy Flying!

To examine the protection scheme, load in a monitor that resides at \$8000 or above. Read in the file BOOT with the monitor. It starts at \$0900. Use the I command (HIMON or LOMON) or M command (HESMON, MICROMON) to display memory from \$0900 to \$0990. You should see the message "HARDWARE FAILURE - CHECK DISK DRIVE". This is the message you would get if you tried to run the backup copy without the above fix. Note the starting address of the message is \$096E. Now disassemble the code at \$0991. We see standard routines for loading files. So what takes us to these routines at \$0991?

Disassembling the code above our HARDWARE FAILURE message reveals a BEQ \$0991 at \$092E, occurring after a JSR \$033E. Looking past \$092E we see a routine that clears memory and at \$0951 starts writing the failure message from \$096E. Now we have found out how to get to the file open routines. If we change the BEQ at \$0991 to a BNE we will bypass the FAILURE routines and perform the loading code. By changing the \$F0 61 (BEQ \$0991) at address \$092E to \$D0 61 (BNE \$0991), the program protection is now bypassed.

Looking at the disk directory with a T/S editor shows the program BOOT begins at track \$11 (17), sector \$01. Since our fix is only \$2E bytes into the program (\$092E minus \$0900), the starting sector contains the code we need to change. Add \$04 to our offset of \$2E (2 bytes for the T/S link and 2 bytes for the program loading address contained in the first sector of a program file). The result is offset \$32 into the sector, which should locate the \$F0 61. Changing the \$F0 to \$D0 with a T/S editor quickly removes the protection.

The program is **ADVENTURE MASTER**, Copyright 1984, Christopher Chance, CBS Software.

TYPE OF PROTECTION: The program requires an error 23 on TRACK 18/14.

HOW TO COPY: You may use OMNICLONE to copy this program or any program that will place an error 23.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS:

- 1). Make a copy of the original disk without errors.
- 2). We will capture this program once it is in memory and has passed its protection scheme. The program that contains the protection is the file called "CBSA". This is an interesting program because of its location in memory. The program is stored from \$0300 through \$7FFF. If we attempt to load the program through a monitor, it will take control of the computer.
- 3). Scratch the CBSA file from your copy disk so that you can make room for the new program.
- 4). Load the program from the original disk. When the ADVENTURE MASTER menu screen appears, reset your computer.
- 5). Load and execute LOMON with SYS32768. Save the new program with S "CBSA",08,0800,7FFF.
- 6). Now for the entry point. Another place to look for an entry point is a store (STA) to screen or background color. We will begin this search with the hunt command. Using H 0800 7FFF 8D 20 D0 will return six memory locations. The first address returned is \$4806. When we investigate that location with the D command, we do not find a load instruction (LDA). Before you can store something, you must have a value to store. \$4806 would appear to be a dead end. Let's try the next location which was \$4821. When we investigate this area with the D command, we find our load (LDA) instruction at \$481C. Through this code, the color black is being stored in the border and background color locations. With your copy disk in the disk drive give this entry point a try with G 481E. That will do it.
- 7). To use this program LOAD "CBSA",8,1. To activate use SYS18460. If you are fond of the CBS screen, activate the program with SYS18432.
- 8). YOU'RE DONE.

The program is **BANK STREET MUSICWRITER**, Copyright 1985 MINDSCAPE, INC., Copyright 1984 by Glen Clancy. By the way this is probably one of the best music program for the C-64 that we have seen!

TYPE OF PROTECTION: This program utilizes non-standard sectors as its protection scheme. An investigation of TRACK 19 reveals a duplication of SECTOR 3 in place of SECTOR 18. You will find non-standard sectors explained in greater detail in the Program Protection Manual II.

HOW TO COPY: Some of the newer nibble copy programs will copy this disk, such as DISKMAKER.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

We will unprotect this program by lifting a working copy of the program from memory. It is faster than tracing through the protection scheme and will cut down on the load time.

1. Load and run "FILL'ER UP". Have a blank formatted disk available.
2. If you do not have "FILL'ER UP", LOAD and execute HIMON with SYS49152. Once executed, fill memory with F 0800 BFFF 99. Exit to BASIC with G FCE2 (SYS64738).
3. Load and execute the original program in the normal manner. Once the program is in memory (i.e. main menu), RESET your computer.
4. Load and execute HIMON. Using the I command, we find that the program code extends from \$0800 through \$95E1. Save this code to a formatted disk with S "BANK STREET",08,0800,95E1.
5. With that accomplished, we will now search for the entry point. Using the D command at \$0800, we find the first meaningful code at \$0850. As we have explained in the PPM II, a good place to try an entry point is at a JMP instruction.
6. Place the original disk in the drive and try the entry point with G 0850. This section of code loads the program called "SPRITES.BIN" and jumps to the routine that displays the menu screen (\$08CD).
7. Use a file copy program to copy "SPRITES.BIN" to your formatted disk. You may also wish to copy the programs from SIDE TWO of your original disk to your formatted disk.
8. To utilize the program, LOAD "BANK STREET",8,1. Once in memory, execute the program with SYS2128 (HEX \$0850).
9. YOU'RE DONE!

The program is **BANK STREET WRITER**, (c) 1983 by The Bank Street College of Education.

TYPE OF PROTECTION: The program checks for an error 27 on track 34 and an error 23 on track 2.

HOW TO COPY: Most copy programs will copy this disk.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

1. Make a copy of the disk without any errors on it.
2. Load HIMON and execute it with SYS 49152. Fill memory using F 5000 5E00 99.
3. Load OBJ5000 with L"OBJ5000",08. Display memory with M 5530 and change the first byte from an \$A9 to a \$60.
4. Save out the altered code to the COPY disk with S"@0:OBJ5000",08,5000,5E00.
5. That's it - you're done!

Let's take a closer look at this program. BSW is the boot program. It is written in BASIC and may be loaded with ",8". After loading the program, take a look at lines 1500 through 1680. This is the error checking routine. The variable NR is used to compare the value returned from the error channel. Now the problem is that this boot is used to load two different modules. The word-processing module is looking for an error 23 and the tutorial program checks for an error 21. Changing the value of NR from a 0 to a 1 would allow the tutorial program to pass the protection check, but the word processing module would crash. As you can see, changing the boot program will not solve our problem.

The boot program loads in a series of machine language programs. Once these programs are loaded, the program will begin execution at 6144 (HEX 1800). This process is also revealed in the boot program. If you load in "OBJ1800" through HIMON, you will find that the second subroutine is a JSR to \$5535. This is the code that leads to the error checking. We eliminated that check with our 60 (RETURN SUBROUTINE). Following the code at \$5535 reveals a JSR to \$6800. If you load "OBJ6000" and check the code, using the M command, you will be able to spot the U1 command. Our change will eliminate the check entirely, allowing both modules to execute properly from the unprotected disk.

For those who have purchased the EPROM PROGRAMMER'S HANDBOOK and wish a cartridge version of this wordprocessor, follow the procedure below:

1. LOAD "MICROMON 49152",8,1 from your EPROM UTILITY DISK and activate with SYS 19152. Use F 0800 BFFF 99 to clean up the work space. Exit to BASIC with G FCE2.
2. Load the word-processing module from the unprotected disk in the usual manner. Once the program is running, RESET your computer.
3. We will now add a BASIC line to the program. Type 10 SYS 6144 and press RETURN.

4. Re-enter the monitor with SYS 49152. Save out the code, to a formatted disk with S"BANK STREET",0801,7B61.
5. To burn the chips, follow the procedure outlined in chapter 18 of the EPROM PROGRAMMERS HANDBOOK. As you know, this chapter describes the process for burning the Consultant (tm) and Paperclip (tm) to a bank switch board. If you prefer Bank Street Writer to Paperclip, you may burn this program to chips 2 and 3. Just follow the instructions given on Page 126 of the Eprom Handbook. Don't forget to use Micromon's relocate command to load Bank Street Writer at \$1000 (L 1000 "BANK STREET").

If you prefer to work with a single file copy of this word processor from disk, follow STEPS 1-4 of this procedure. HIMON will do the trick in place of MICROMON. To activate the program, use LOAD "BANK STREET",8 and RUN.

The program is BC'S QUEST FOR TIRES (tm), COPYRIGHT 1983 BY SIERRA ON LINE INC.

TYPE OF PROTECTION: The auto start routine first blanks the screen so that no characters may be seen. Then it loads a user file from the disk into the screen memory. The program then jumps to the memory contained at the screen location (\$0400). This code will perform the necessary error checks (bad blocks) and then loads the main program from the disk. The main program is stored on the disk in user files and must be loaded back into memory in the proper order and at the proper location. If the proper error is present then the program will execute.

HOW TO COPY: Use any good copy program to copy the disk and place the appropriate errors on the copy disk. One of the newer nibbler copy programs may also be used to make a working copy.

HOW TO MAKE A WORKING COPY WITHOUT ANY ERRORS ON THE DISK.

The technique used here is similar to that illustrated in pages 50-60 of the Program Protection Manual. We will also use one of the special utilities from the disk call U1 & U2.

- 1) Copy the disk with any good copy program that does not place any errors on the copy disk. The copy program must copy all the data from the disk (remember that user files may be placed any where on the disk).
- 2) Disassemble the loader program using U1 & U2 (from track 17, sector 0). This program loads a user file from the disk and stores the code at \$0400. To find out what block of memory will be loaded, use the 'I' command of your ML monitor. You will find the following ASCII code 2 81 0 2:1U. Try reversing the order of the code to find out the proper block (U1:2 0 18 2), track 18 sector 2.
- 3) Disassemble the USER file (track 18, sector 2) using U1 & U2. Comment the code (see pages 52-56 of PPM). Starting at the 10th line of code you will

find the program opens a file for input (JSR \$ffc6), inputs a character (JSR \$FFCF) and then compares this character to the hex value of \$32 (CMP #\$32).

- 4) The code has enough components of a program protection scheme (see chapter 9) to qualify as one. In order to 'fix' this program from beating your disk drive to death, one only has to change the comparison from CMP #\$32 to CMP #\$30. Save the USER file back to the copy disk.
- 5) YOU'RE DONE.

The program is **BEACH-HEAD II**, by Roger and Bruce Carver. Copyright 1985 by Access Software, Inc.

TYPE OF PROTECTION: This program uses a particularly tough type of protection on track 35, not just simple errors.

HOW TO COPY: Some of the new parameter copiers can copy this disk.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

1. Make a copy of the disk without any errors.
2. Load in a track/sector editor. Read in track 17, sector 11 and make the following changes. Note that the byte locations are in hex and start at \$00 in the sector, not \$01:

BYTE	FROM	TO
\$0B-10	20 A5 F6 20 4D F7	EA EA EA A9 AB EA
14-16	20 4D F7	A9 AD EA
1A-1C	20 4D F7	A9 AF EA

Save the modified sector out to the copy disk.

3. That's it - you're done!

The program uses a subroutine at \$F74D (in the RAM under the KERNAL) to check the protection on the disk. The key protection value is left in the accumulator when it returns from the protection routine. This key value is saved in memory. The protection routine is called two more times and each value is also saved. Once these values are placed in memory, the program will run. The modifications we made up above disable the calls to the protection check routine. We simply place the correct values in the accumulator each time, and they are stored in the proper locations as usual.

A word of explanation is in store for those of you that have been trying to ISEPIC this program using the instructions in the August Newsletter. Those directions were given to us by Chip Gracey, the designer of ISEPIC. It turns out that the process is considerably trickier than it seems from his explanation. The

program must be ISEPIC'ed at a particular moment, and then you still need a nibble copy of the disk for the rest of the program code. We think you'll find that the method we give you this month is adequate for archival purposes.

This program uses its own alternate communications routine located at \$F74D in the RAM under the KERNAL. You might want to examine this area of code under the KERNAL, it could prove interesting!!

The program is **BELOW THE ROOT**, copyright 1984 by Windham Classics.

TYPE OF PROTECTION: Track 2 is checked for an error 23 and track 3 is checked for an error 27. The errors can be anywhere in sectors 1-16.

HOW TO COPY: Any good nibbler will copy this disk.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

1. Make copies of BOTH SIDES of the disk with no errors on them.
2. Load in a track/sector editor.
3. Insert SIDE 1 of the COPY disk in the drive. Read in track 14, sector 3. Change the byte at location \$20 (32) from the value \$20 (32) to the value \$4C (76). Write the sector back to the COPY disk. That's it - you're done.

This program is on the easy side, but not trivial. The boot program is a BASIC program called WIND. It loads in most of the other files on the first side, and then does a SYS 36864 (\$9000). We traced the code at \$9000 for a bit, then decided to try a more direct approach. When you suspect that simple bad blocks are being used (as evidenced in this case by the TWO head "bumps" when the original is loaded), one classic technique is to look for a "U1" command. Using a monitor at \$C000, we loaded as many of the files into memory as possible. Then we flipped out BASIC (change location \$0001 to \$36) and hunted through the code for a U1 command using H 0800 BFFF "U1". Note that the command to hunt for an ASCII string varies slightly from one monitor to another. You can also hunt for the same bytes in hex: H 0800 BFFF 55 31.

Sure enough, we found a U1 command - two in fact. One was at \$80ED (in the file "GAME") and the other at \$3833 (in "GAMELOW"). We looked at the one at \$80ED first, since it was in the same file that the BASIC boot SYS's to. To make a long story short, looking into this code did not reveal any check for errors. We concluded that these routines were part of the normal operation of the program, probably for loading and saving games (this is an adventure game). Fortunately, the other U1 command at \$3833 turned out to be the one we were looking for.

To find where the U1 was used, we hunted for a reference to the beginning of the U1 command: H 0800 BFFF 33 38 (remember, two-byte addresses are usually stored in reversed order - lo byte/hi byte). This turned up one reference at

\$37E9. Bingo! Backing up a bit, at \$37D0 it converts the sector number (in the "Y" register) from hex to ASCII and stores it into the U1 command. Then at \$37E1 it sends the U1 to the drive. At \$37F6 it uses the KERNAL routines TALK (\$FFB4) and TKSA (\$FF96) to open up the error channel (15) for input (see "Alternate Kernal Routines" in the Aug. 85 Newsletter). These routines are much less common than the usual KERNAL routines, and this program is a good example of how they're used. Next, the error number is input using ACPTR (\$FFA5), another alternate routine. If the error matches the one stored at \$3840-41, the routine returns where it came from. If not, the sector number (in "Y") is incremented and the U1 is done again. It keeps trying until sector 16. If it still hasn't found the right error, it pulls its normal return address off the stack. Then it does an RTS, which returns it to the PREVIOUS calling place. This skips an important routine at \$3B5D. We've modified the code so it always returns properly.

Thanks (and a \$50 check) to Alan O'Dale from way Down Under for the following:

The program is **BLITZ! BASIC COMPILER**, Copyright 1983 by Prolic, Inc.

TYPE OF PROTECTION: The disk contains many errors but the program only checks for an error 22 on track 35, sector 4.

HOW TO COPY: Use any copy program which can copy error 22's.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

1. Load in a monitor at \$8000 or above, such as HIMON.
2. Insert the original disk and load the compiler program with L "COMPILER",08
3. Change location \$4243 from \$42 to \$40.
4. Save the program to a pre-formatted disk with S "BLITZ!",08,0801,4E9C.
5. That's it - you're done!

About all we can say for this is - it works! The modified program can be put on any disk; it contains the complete compiler. By the way, the original BLITZ! disk contains many programs on it besides the compiler. The other programs have all been scratched, but many can be recovered by un-scratching' them and then file copying them from the disk (see PPM Vol. I for information about un-scratching files). The extra programs appear to be demo programs for some old plotter model. Who knows what other buried treasure lurks on your commercial disks?

The program is C POWER V2.6 (tm) by Brian Hilchie. Copyright 1985 by Pro-Line Software Ltd. This is a "C" language system for the C64.

TYPE OF PROTECTION: There are many errors on this disk, especially tracks 33-35. The errors are not checked directly, however. A special process is used to read the information "under" the block at 34/6.

HOW TO COPY: Some nibble copy programs may be able to copy this disk.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

1. Make a copy of the disk using a 3-minute type copy program which DOES NOT put errors on the disk.
2. You'll need a disk monitor such as DRVMON from DI-SECTOR. If you have a different disk monitor, you'll have to translate the following DRVMON commands to the proper ones for your monitor. (see step 7 for another method)
3. First, load in and execute DRVMON. Switch over to editing drive memory with the command: 08 (Letter O, digit 8). The 8 is the device number. Insert the ORIGINAL disk and initialize the drive with: @IO .
4. Now we need to try to read block 35/6 from the original disk. To do this, we'll put a "read" job command and the track/sector numbers into the job queue of the drive. Display locations \$0000-0007 with: M 0000 0007. Move the cursor over to the first byte displayed and change it to \$80 (read job). Now move to the last two bytes on the line (\$0006-0007). Change them to \$23 06 (track 35, sector 6). Hit RETURN. After the drive finishes, display location \$0000 again. You should see an \$03 error return code there. This indicates a DOS error 21 - no SYNC mark. Although we didn't read any information, this procedure has positioned the R/W head for our next operation.
5. Now we want to read in block 34/6 from the original disk. Follow the same procedure as in the last step except use \$22/6 (34/6) for the track/sector (DON'T initialize the drive!). The error code returned in \$0000 should be \$01, which means no error occurred. The sector was read into buffer 0 at \$0300-03FF. Display this area with: M 0300 03FF. Around location \$0370 you should see the message "Illegal Function Call".
6. Remove the original disk and insert your copy disk. Now we're going to save the buffer of information onto block 34/6 of the copy disk. Display location \$0000 and change it to \$90 (write job code). Hit RETURN. Again the error code returned in \$0000 should be \$01 (OK).
7. That's it - you're done. You can copy the disk now with any 3 minute copier, but don't try to file copy the disk. You should test your copy to make sure it works.
8. You can use the EPYX FASTLOAD (tm) cartridge's built-in track/sector editor to do the above procedure too. Just read block \$23/06 and then block \$22/06 from

the original disk. Insert your copy and write the sector to \$22/06. Other T/S editors may not work (DI-SECTOR and CLONE MACHINE did not work for us).

Okay, you're finished, but what in the world have you done? Well, apparently block 34/6 of the original disk contains information written "under" the error in that block. By trying to read 35/6 first, the drive was left in the "correct" abnormal condition. Once the information in block 34/6 was read successfully, it could be written back to the copy disk.

The compiler is the only program on either side of the disk that is protected. The back side contains library routines, and you should copy it too with a 3-minute copier. Testing your backup of the first side to make sure it works requires compiling a program. For those of you that aren't familiar with the "C" language, here's how to test your backup.

First, you shouldn't use the backup you just made for this test since it involves writing to the disk. If something went wrong, you might have to repeat the backup procedure. Instead, start by making a 3-minute type copy of the BACKUP disk. Now execute the C SHELL program with: `LOAD":*"/8` and `RUN`. We'll test the compiler by compiling a C program which is already on the disk. The program source code file is called `PRINT.C`, and it's already been compiled into a file called `PRINT.SH`. This program just prints a listing of a program onto the screen or printer.

When you get the "\$" prompt, scratch the compiled version of the program with the `C REMOVE` command: `RM PRINT.SH`. You can list the disk with the command `L`. Now run the compiler on the source code with the command: `OC -P PRINT.C` (if you leave off the `-P` it will prompt you for the source, object and compiler disks). You'll see the source code listed as it is compiled. If the copy is bad, the compiler will usually freeze up at the first `IF` statement it comes to. To be sure, you should finish the whole compile procedure.

To do this, load in the linker with: `LINK`. At the link prompt `>`, type in `PRINT.O` and hit `RETURN`. `PRINT.O` is an intermediate file that was created by the compiler. When it's finished reading the file in, put in the BACK SIDE of your C copy disk with the library routines on it. Type an `UP-ARROW` and hit `RETURN` to read the library routines in. Now insert the SECOND copy disk (the one you've been working with) and press `RETURN`. When it prompts you for a file name, type `PRINT.SH`. After a bit you should be returned to the "\$" SHELL prompt.

To test the newly compiled `PRINT` program, use the `PRINT` command to list its own source code. Type `PRINT PRINT.C` and hit `RETURN`. You should see the same code listed that you saw earlier in the compile stage. If all is well, you can scrap the second backup copy and just save the first one.

The same procedure we used on `C POWER` (tm) reportedly works on another Pro-Line program, `WP64` (a word processor). We haven't tested this, however.

The program is **Championship Boxing**, Copyright 1985 by Sierra On-Line.

TYPE OF PROTECTION: The program checks for an error 23 on track 18 sector 18 during the initial load, then checks for the same error again after loading in the individual boxer's info (before the fight starts). This makes a memory copy ineffective because you would have to Snapshot every possible combination of some 30 or more boxers!

HOW TO COPY: Any program capable of reproducing an error 23 will copy this program, or you can fast copy it and place on error on track 18 sector 18 with an error maker.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

- 1) Make a copy of the original disk without errors. I now use FAST HACK 'EM single or double (unprotected) backup...FAST! Beware that some versions of 4-MIN copy will reproduce the error 23!
- 2) Use a monitor that does not occupy 'C' block (49152). LOMON or LIMON will do.
- 3) Load in the program GRAFX. Disassemble the code starting at \$C191 (D C191). Change the CMP #\$32 at \$C191 to CMP #\$30. Change the CMP #\$33 at \$C198 to CMP #\$30. Save back to disk with S"@0:GRAFX",08,C000,C202. Use the proper syntax for your monitor, if different.
- 4) Now load in INTX. Disassemble the code starting at \$C686. Change the CMP #\$32 at \$C686 to CMP #\$30 and the CMP #\$33 at \$C68D to CMP #\$30. Save back to disk with S"@0:INTX",08,C000,C6F7.
- 5) You're done!

EXPLANATION:

The code you changed with the monitor causes the program to look for no error after reading track 18 sector 18, which should be true on your back-up disk. This program is similar to several others in it's checking for a simple error on the directory track. Although I figured this to be an easy program to de-protect (and really it is), it still had me working on it for a couple of days. I didn't realize there are TWO separate programs doing the error checking! I found the error checking routine in INTX quickly, and changed it...but the program continued to 'crash'. I examined the code further to see what I'd missed, but that wasn't the problem. The problem was that GRAFX also loads into the same area, at a different time, and does its own little check! Things are not as obvious as you may think sometimes! After altering both routines, the program works without errors. You might also note that a big portion of the program is in BASIC and may be easily LOAded and LISTed.

The program is COHAN'S TOWER, Copyright 1983, by Datamost, Inc., Copyright 1983, Fanda, Licensed by Datamost.

TYPE OF PROTECTION: This program will check track 02/sector 09 for an error 23 and track 02/sector 10 for an error 20. If present the program will load normally. (Notice that this program will check two (2) bad blocks!! They make your read/write head beat twice against the end stop. Can you believe that?)

HOW TO COPY: Make a copy of the disk and place an error 23 at track 02/sector 09 and an error 22 at track 02/sector 10. You may also use one of the newer copy programs that will place the errors for you.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

- 1). Load the program from the original disk with LOAD":*",8,1.
- 2). Once the program is running, insert your cartridge-based machine language monitor with the cartridge switch off. You may also use the Cardco 5 expansion board with the left switch on. Reset your computer with the cartridge switch off.
- 3). You should now have the familiar blue screen, showing 30719 BASIC BYTES FREE.
- 4). Now turn on your cartridge switch (Cardco - right switch) and hit the RESTORE key. You should now be in your cartridge monitor.
- 5). Since we had to use the "back door" for this program, we must now find an entry point. The best place to start is a store (STA) to border (D020) or background (D021) color. We will use the hunt command to begin our search. H 0800 7FFF 8D 20 D0 and press return. Six memory locations will be returned by the computer.
- 6). We'll use the first location \$0918. When we scroll through this section, we find stores to border and background. This spot looks promising. Let's give it a try.
- 7). Since we have to load (LDA) something before we can store (STA) it, we will begin at \$0916. Use G 0916 to execute this section. As you can see, we are back in the game.
- 8). We will now save out the code to a formatted disk. Repeat steps 1-4 to regain control of the program. Once you are back in the cartridge monitor, save the code with S "COHAN'S TOWER",08,0800,7FFF. To utilize the program, LOAD "COHAN'S TOWER",8,1 and activate with SYS2316 (HEX 0916), or write a boot program to activate the program for you.
- 9). YOU'RE DONE.

Let's do a little more investigating with this program. Repeat steps 1-4 with the original disk. Once you are in the cartridge monitor, load and execute

HIMON, through the monitor with, G C000. Turn off your cartridge switch and remove the cartridge monitor. Using the I command, scroll through the code at \$8000. You will see a CEM80. This is usually the first place we look for an entry point. This usually means that the program will emulate a cartridge start. This is a CEM80 with a twist. As you have learned from the P.P.M. and the NEWSLETTER, the warm-start vectors are located at 8002 and 8003. This would make our starting address \$8009. Let's execute this code with a G 8009. Now reset your computer, as described, and examine the code with the I command. As you can see, there is nothing left to interpret. Our program code has been replaced with BRK's. This was the purpose of the code at \$8000.

NOTE: DON'T GIVE UP AFTER ONE DEFEAT! VICTORY MAY ONLY BE A FEW BYTES AWAY.

The program is Copy-Q II, Copyright 1985 by Q-R&D.

TYPE OF PROTECTION: This program uses different types of non-standard formatting as well as half-tracking.

HOW TO COPY: This disk is very difficult to copy; we could not find a copy program which would work consistently.

HOW TO MAKE A WORKING COPY WITHOUT ANY ERRORS ON THE DISK:

1. The original disk contains 'templates' which are used to copy different programs. The 'templates' are not in files, but are stored on tracks 1-17 in various non-standard ways such as on half-tracks. The first thing to do is make a copy of the original disk, using Copy-Q II to copy itself. This copy will not work, but the 'templates' will be copied correctly. We'll modify the Copy-Q II program and put it on the copy disk with the templates.
2. We need to mark the first 17 tracks 'used' in the BAM so when we add the Copy-Q II program it will not use these tracks. Load a track and sector editor and run it. Read in block 18/0 from the COPY disk. Change bytes \$04-47 to all \$00's. If your T/S editor has an ML monitor, you may prefer to use its 'fill' command to change these bytes. Write the block back out to disk. Exit the T/S editor back to normal BASIC. (Note: bytes are numbered starting at \$00, not \$01)
3. Load a monitor at \$8000 or higher, such as HIMON, but do not execute it. Type 'NEW'.
4. Insert the ORIGINAL disk and type LOAD "MENU",8 After the menu loads, type POKE 2931,96. This puts an RTS at \$0B73 in place of JMP \$0900.
5. Now RUN the menu program. The main program will load in about 30 seconds, but the RTS at \$0B73 will cause it to crash. You'll still be able to enter commands, however.

6. Execute your monitor, which should still be intact. The Copy-Q II program is now in memory at \$0800-3FFF. We'll preserve a small section by transferring it up in memory: T 0800 080F 4010
7. Next, we'll write a small piece of ML to transfer this memory back when the program executes. Assemble the following code at \$4000:

```
4000 LDY #$0F
4002 LDA $4010,Y
4005 STA $0800,Y
4008 DEY
4009 BPL $4002
400B JMP $0900
```

8. Now we need to put the BASIC statement 10 SYS 16384 at the beginning of the program. This will jump to our transfer routine above (\$4000=16384) and allow us to execute the program with a RUN command. Type M 0800 080F and enter the following bytes:

```
:0800 00 0D 08 0A 00 9E 20 31
:0808 36 33 38 34 00 00 00
```

9. Insert the COPY disk and save the program with
S "0:COPY-Q",08,0801,4020
10. That's it - you're done. The program can now be LOAded and RUN like a BASIC program.

The program is COUNTDOWN TO SHUTDOWN, Copyright 1985 by Activision.

TYPE OF PROTECTION: The disk is heavily protected with nonstandard formatting.

HOW TO COPY: Some of the new parameter copiers can make an archival backup of this disk.

HOW TO MAKE A WORKING COPY WITHOUT PROTECTION ON THE DISK:

1. Make a copy of the disk with a copy program that doesn't put errors on the disk.
2. Load in and execute a track/sector editor.
3. Read in track 17, sector 7 from the copy disk.
4. Change the following bytes. Note: byte locations in the sector are numbered starting with \$00 as the first byte.

LOCATION	CHANGE TO
\$2A	\$BB
C1	80

5. Save the block back out to the disk (at track 17, sector 7).

6. That's it - you're done.!

This exact same set of changes will work on several other Activision programs, such as FIREWORKS CONSTRUCTION SET. To tell if these changes might work on a particular program, look for the same code in block 17/7 as on COUNTDOWN. Make the changes given above and try it. The code might also be located in a different block.

The program is CREATIVE FILER (tm), Copyright 1984 by Creative Software.

There are at least two versions of the protection scheme for this program. The newer version has two files called LOADER and X1. If yours has two files called CF and TI, you have the older version. We'll cover both of them, starting with the newer, more interesting one. In fact, the new version is such a good learning tool that even if you don't like the database program itself (a distinct possibility), you should buy the disk for the protection scheme.

NEW VERSION:

TYPE OF PROTECTION: The original disk contains special information written with altered density on track 36. Also, the protection code is encrypted.

HOW TO COPY: Some of the new nibble copy programs that can handle altered bit density and extra tracks will work.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

1. Format a new disk with the name "CREATIVE FILER" (any ID will do).
2. Load a track/sector editor and run it.
3. Read in block 18/0 from the NEW disk. Change byte \$50 from \$13 to \$12 and byte \$52 from \$FF to \$F7. Write the block back out to 18/0.
(Note: bytes are numbered starting at \$00, not \$01)
4. Read in block 20/11 from the NEW disk. Change bytes \$00-05 to \$A9 FF 8D FF 01 60. Write the block back out to 20/11.
5. File-copy all the files from the original disk to the new copy.
6. That's it - you're done.

This is practically a 'textbook' protection scheme, well worth examining in depth. There is nothing startlingly new about it, but it does contain an effective custom DOS routine combined with good use of encryption. If you have PFM Vol. II, you'll find similar routines covered there. The protection code is contained in three places: in the file "X1", in block 20/11 and on track 36. The boot program "LOADER" loads and executes the main protection routine "X1" at \$7000-74A9. "X1" starts by altering the character set, setting some sprite data and printing the title screen. Then it adds up a section of itself (\$7278-BE) using ADC (ADD WITH CARRY). The resulting one-byte checksum (\$A7) is then EOR'ed (exclusive-or'ed) with most of the rest of the code (\$72BF-74A7) in order to decrypt it. Actually, only part of the code is fully decrypted at this point; the rest will be EOR'ed again later.

Next, a major subroutine at \$731C is called. It starts by opening up a random access file (#) and reading in block 18/0, through a U1 command at \$7489-9A. This block contains the BAM and disk name. Then the routine sets the buffer pointer to the start of the disk name, using a buffer-pointer (B-P) command at \$749B-A6. It reads the disk name into \$6002-0F and then compares it to the name "CREATIVE FILER", restored at \$7004-11. If there is any mismatch, the routine crashes. The routine uses the KERNAL routines LISTEN (\$FFB1), CROUT (\$FFA8) and UNLISTEN (\$FFAE) to send the "U1" and "B-P" commands. This can be a form of protection since these routines may not be as familiar as the normal CHROUT (\$FFD2) routine.

Having passed the first protection check, it really gets down to business. It opens up another file to the drive, this time specifying buffer #1 (\$0400-FF). As explained last month in the article EXECUTING 1541 ROUTINES, some custom DOS routines must be loaded into a particular place in memory, and this can be done by specifying the buffer # rather than letting the drive pick one for you. In this case, a DOS routine is loaded from block 20/11 into buffer #1 and executed, all in one step. This is done with a handy block-execute (B-E) command at \$7054-66. Like many custom DOS routines, you don't have to know what this routine is actually doing, just what value is returned eventually. This DOS routine is interesting enough that we'll take a look at it anyway. Remember, this routine is executed in drive memory.

The first thing the custom DOS routine does is disable interrupts (SEI) so it has total control of the drive. Next, a small routine decrypts the rest of the code (\$0412-F1) by EOR'ing it with the value \$AC. Then the decrypted code is executed. It initializes the R/W electronics (JSR \$0461) and sets up two track pointers. These pointers specify that the head is currently on track 20 (from the B-E command) and that it would like to go to track 36 next. A subroutine at \$0471 moves the head based on these pointers and waits a bit for the head to settle in. This subroutine also sets the proper bit density (clock speed) for the track being moved to. This is a very handy routine! After returning to the main DOS routine, the density is reset to the value for track 26. The head is now on track 36 ready to read some information at track 26 density.

The program code at \$0432-3E waits for a sync mark to come along (using a subroutine at \$04E6-F1) and then waits for a special marker byte, \$5C. After the

\$5C is detected, the routine reads the next 256 bytes into drive memory at \$0500-FF, and then 69 more bytes into the OVERFLOW BUFFER at \$01BB-FF. A subroutine in the normal DOS ROM at \$F8E0 is called next. Recall that raw data from the disk is in a special code called GCR, not in hex. The routine at \$F8E0 is normally used to process a header block from disk. It converts the bytes in the OVERFLOW BUFFER from GCR to HEX, saves the first hex byte (header block id) separately and puts the rest of the hex into a regular buffer, in this case at \$0500. After this subroutine finishes, the new code at \$0500 is executed with a straight JMP. This code leaves an \$FF byte at location \$01FF in drive memory and terminates the block-execute command (finally!). The disk protection has now been checked - the \$FF value at \$01FF is the key to the protection scheme.

Meanwhile, back at the computer... after issuing the B-E command, the code in the computer sends a memory-read (M-R) command to read the value left at \$01FF in drive memory. The value received is stored at \$70A8 in the computer. If this value isn't equal to \$FF, the program crashes. Otherwise we're over the hump, and it's all downhill from here (with a few detours). The next part of the routine adds up the section of code from \$731C-74A4 (using ADC) to get a one-byte sum (\$8D). This value is used to EOR two sections of code, \$72BF-744B and 74A5-A7. This is interesting because all of this memory was already EOR'ed earlier. The net effect is to finally decrypt the areas \$72BF-731B and \$74A5-A7, and RE-ENCRYPT the rest of it! At this point, we have come to the end of the major subroutine. We return back to the main routine at \$72BF, an area which was just decrypted. This last section of code looks a little involved, but it simply loads in the files "CODE", "MATH", "BCDF" and "GOGO" at their normal locations. Finally, all open files are closed and the two bytes at \$8000-01 (cartridge cold start vector) are used as an indirect vector to start the main program.

To summarize the protection method, the disk name is checked and a custom DOS routine is executed which leaves a key value in drive memory. We take care of the disk name in step 1 of our directions for removing the protection. The rest of the steps are concerned with the B-E routine. In step 2, we modify the BAM to mark block 20/11 as used, so the regular files won't wipe it out. Then in step 4, we put a short routine into 20/11 which simply stores our key value into drive memory and returns. The disassembly of this routine is:

```
0400 A9 FF    LDA #$FF
0402 8D FF 01 STA $01FF
0405 60      RTS
```

This routine replaces the custom DOS routine, and thus makes the special data on track 36 unnecessary. After copying the regular files you are all set. Other files can be added to this disk safely, as long as you don't validate the disk.

OLD VERSION:

TYPE OF PROTECTION: The program checks track 2, sector 0 for an error (any). The disk name is also checked.

HOW TO COPY: Use any copy program which can copy errors, such as Omniclone.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

- 1) Copy the disk with any copy program which doesn't put errors on the backup disk. You may file copy the disk if you prefer.
- 2) Load and run HIMON. Fill memory with F 0801 7FFF 99.
- 3) Load the protection routine with L "TI",08
- 4) Display memory with M 707D. Change bytes \$707D-7F from \$20 CF FF to \$BD 54 71.
- 5) Display memory with M 7127. Change \$7127 from \$32 to \$30.
- 6) Replace the altered routine with S "@0:TI",08,7000,741E
- 7) That's it - you're done!

This program first reads the disk name and compares it against a prestored value at \$7154. The patch in step 3 above alters the routine so that it compares the prestored value with itself. The program also checks block 2/0 for any error, using the time-honored CMP #\$32. Step 4 changes it to CMP #\$30 as usual so that it passes on an error-free disk. The disk can now be file copied or have other files added to it.

The program is **CROSSWORD MAGIC**, Copyright 1985 by Mindscape, Inc.

TYPE OF PROTECTION: This disk contains a special type of error, not just simple bad blocks.

HOW TO COPY: This disk is difficult to copy. Some parameter copiers can make a working version of this disk.

HOW TO MAKE A WORKING COPY WITHOUT PROTECTION ON THE DISK:

1. Make a copy of the disk with a copy program that doesn't put errors on the disk.
2. Load in and execute a track/sector editor. Read in track 12, sector 6 from the copy disk and change the following bytes.

LOCATION	CHANGE TO
\$8B	\$53
8D	2C

3. Save the block back out to the disk.

4. That's it - you're done.!

The changes given above will disable the protection check. An archival backup copy with no errors on it will now pass the protection check and run properly.

The program is CRYPTO CUBE, Copyright 1983 DesignWare (tm), Inc.

TYPE OF PROTECTION: This program check block 1/1 for an error 23.

HOW TO COPY: Most copy programs will copy this disk.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

1. Make a copy of the disk without errors on it.
2. Load a track/sector editor. Read in TRACK 6, SECTOR 17 and change bytes \$4C-4E from \$A9 00 20 to \$4C D3 08. Write the sector back to your COPY disk.
3. That's it - you're done!

Studying this program posed a few problems. First of all, the code is stored in random blocks on the disk. We could trace the code with a track/sector editor that includes a monitor feature, or we could allow the program to come into memory and reset the computer. Working through a track/sector editor can be a bit cumbersome, so we chose the latter method. The only problem with the reset approach is that the program includes a check at \$8000 that erases the program code on reset. We can get around this problem as follows: After the error check, the screen turns white. At this point you should hold down your RESET button while you ground the EXROM line with a cartridge switch (see PPM Vol I). Once the switch is engaged, release the RESET button. When you are returned to BASIC, load and engage HIMON.

From here, we just scrolled through the code looking for something interesting. We found a U1 command at \$80E0. Using the D command, disassemble the code beginning at \$804C. This is where we find the files being prepared for the U1 command. At \$80CA, the problem begins. The \$FFCF KERNAL call (CHRIN) inputs a byte from the error channel. This value is AND'ed with #\$0F, which eliminates the upper nibble of the value. The new value is now pushed on the stack (PHA) and we go back and wait for a RETURN character (ASCII \$0D). If we continue the disassembly through \$80F3, we find a jump to \$0800. This is the tip-off. If you check out the code at \$0800, you will find that it has been altered. The first byte was altered by our reset, but the rest was done through the program. The code at \$8000 has also been changed. The first good piece of code is a JMP to \$08D3. The original code at \$0800 contained a JSR to \$8000, where the start-up message was checked and a return was executed. From there, it jumps to \$08D3 where the program continues normally.

It turns out that we can skip the error routine entirely by slipping a JMP \$08D3 directly into the code. We will insert that jump at \$804C, which is where the error routine begins. Easy solution except for one slight hitch. As we pointed out earlier, the program is stored in random blocks on disk. (No one said it would be easy). The next step is to dig out your track and sector editor and read in each block of the disk until you find the U1 (or write a BASIC program to do this). You will find this code on track 6, sector 17. Now you must locate the beginning of the error checking routine, which will eventually be stored at \$804C. This section of code begins with \$A9 00 and may be found at byte \$4C in the sector. This is where we inserted the JMP \$08D3 in Step 2 of the unprotection procedure.

The program is **DECATHLON**, COPYRIGHT 1983-84 ACTIVISION.

TYPE OF PROTECTION: This program checks track 18 sector 9 for an error 23 CHECKSUM ERROR IN DATA. If this error is present, the program will run properly.

HOW TO COPY: Copy the disk with any good copy program and place an error 23 on the disk at 18/09. You may also use a nibble copy program and let it put the errors on for you.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK.

We will combine many of the techniques illustrated in the P.P.M. to unprotect this program. The problems we will encounter are as follows:

PROBLEMS:

- 1). Getting past the error checking.
- 2). The program resides in cartridge memory, producing an auto-start and preventing a normal reset.
- 3). A portion of the program resides under basic.

SOLUTIONS:

- 1). Let's get past the error protection first. LOAD and RUN the program. With your cartridge switch (P.61) off, insert your cartridge based machine language monitor. Reset your computer with your switch off. This will return us to basic. Remove your cartridge monitor.
- 2). LOAD and execute LLMON with SYS 8192.
- 3). Using the M command, go to location 0001 and change the 37 to a 36. This will flip out the basic interpreter, allowing us to access the code underneath. .:0000 2F 36 20 3E 32 00 23 00

- 4). S "DECATHLON",08,8000,C400
- 5). LOAD"DECATHLON",8,1
- 6). Press RESTORE to activate the program.
- 7). YOU'RE DONE.

This is a good time to explain the function of the non-maskable interrupt (NMI). There are three types of interrupts: IRQ (interrupt request), NMI (non-maskable interrupt), and the BRK (break instruction).

The interrupt vector goes about it's work sixty times a second. Among it's chores is updating the clock, checking the RUN/STOP key, handling keyboard input, and other functions necessary for the proper functioning of the computer. With proper preparation, we may change the interrupt vector so that it will suspend normal operations and work on the task we request. Such a request would begin with SEI (set interrupt disable).

The BRK (break) instruction will also suspend the interrupt vector. This instruction is normally used to debug machine language routines.

The NMI is the function that concerns us in this program. On the VIC-20 and Commodore 64, the NMI is used for the RESTORE key and for RS-232 communications. When the RESTORE key is pressed, the computer will suspend operations and proceed to the warm-start vectors stored by the programmer. This is why we are able to interrupt games stored in the cartridge area. Upon power-up or a system reset, the computer will check the cartridge area (8000). If the CBM80 is present, the computer will divert the program to the vectors (entry point) stored there. Remember, the first two vectors are the cold start vectors and the next two are the warm-start. Remember also that these addresses are stored in reverse order. The warm-start vectors are used when the RESTORE key is pressed.

In DECATHLON, a warm start is provided through the program. With LIMON activated, let's take a look at \$8000.

Cold-start	Warm-start	CBM80
55 80	69 80	C3 C2 CD 38

If we load in our unprotected version of the program and enter SYS 32873 (HEX 8069), we will enter the warm-start vector. Fortunately we do not have to remember the address, because the RESTORE key will divert us to this address through the NMI.

This is why we had to use our cartridge based machine language monitor to interrupt this program. When we insert the cartridge monitor and perform a reset, the program sends control to the cartridge. With the switch off, the computer recognizes that a cartridge is present, but will not perform the auto run until the switch is turned on. When the cartridge is removed, the original program code is intact.

Another way to defeat the CBM80 is to search the disk for the CBM80 and change one of the bytes (i.e. change the '8' to '0'). When the reset button is pressed the computer will reset normally. All five bytes of the CBM80 must be present in order for control to be turned over to the vectors. Remember to change the CBM80 back to normal before saving the program to disk.

When run, many programs will alter their code to check the error channel. They will then store these values within the program. If the correct value is not found, the program will not execute properly. These programs are a bit more difficult to unprotect, but with a little effort the result is the same. Programs of this type become easier as we gain experience working with them.

The following was submitted by a newsletter subscriber:

The program is **INSIDE THE COMODORE 64, DEVELOP-64, 4.0 version, FRENCH SILK**, copyright 1983, Don French

TYPE OF PROTECTION: The disk contains an error 23 on track 2, sector 2.

HOW TO COPY: Use any copy program that can copy an error 23.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

- 1) Copy the DEVELOP-64 (D-64) disk with a copy program that will not create errors. (I used the three minute copier that is on my Di-Sector disk.)
- 2) Using a good track/sector editor (I used CLONE this time in the HEX mode) select track 19, sector 0. When you have selected this track and sector, look for two side by side 86's. If you are using CLONE, the first 86 is located in row B, column 8, and the second 86 is located in row B, column 9. Change the first 86 to 84 and the second one to 85, then save the sector back to disk.
- 3) You're done! Neat trick, don't you think? You just changed an encrypted 23 to an encrypted 00.

COMMENTS: On the original disk, they look for an error 23 on track 2, sector 2, and if they fail to find it they crash the program. They were very devious in hiding their error check routine. First they give you a garbaged up directory so that you can't load a couple of files for inspection. One of these garbaged files is an encrypted file containing the 23 error check routine. The rest of the comments will pertain to one of the methods that can be used to locate the error 23 check routine so that it can be modified as it was in the 'HOW TO' section above.

- 1) List the directory from your modified D-64 disk. We will be modifying this disk some more before we're through. You will notice that there are three or four lines of garbage following program "D". These lines will be modified so that we have two programs, one named "BA" and the other "AB".

- 2) Using your track/sector editor, locate the programs named (in hex) "9300" & "0093" on track 18, sector 1. Now change "9300" to "4241" and "0093" to "4142" and save the sector back to the disk. Now list your directory again. The garbage is gone and you have programs "BA" & "AB" in its place. But now, you cannot load your program anymore. Oh well, that's life. Next we will attack autoboot "D".
- 3) You will need a monitor program to play with the autoboot. I use a modified "MICROMON" that is contained on "SYMBOL MASTER" by Schnedler Systems. The reason for this is that it has a couple pseudo ops that allow me to find the starting and ending addresses of programs on disk without loading the programs into the computer. This way I can determine whether I want to use an offset when loading a program with my monitor. Now load autoboot "D" with your monitor using an offset of \$12A7. The boot normally resides at \$02A7-\$0303. (Editor's note: "SYMBOL MASTER" is the excellent symbolic disassembler we discussed in the April issue. It's almost worth its cost just for the MICROMON monitors included. For those of you who don't have a monitor with the offset load feature, simply load in "D" and transfer the code up in memory with T 02A7 0303 12A7).

At location \$12B7 change F7 to FA. This is done so that we no longer look for old "0093", but for the new name "AB" that we are going to locate starting at address \$12FA. Change addresses \$12FA from 00 to 41, \$12FB from 00 to 42, and \$12FC from 00 to 41. Now transfer your program to its normal memory location with T 12A7 1303 02A7. Save the new "D" to your disk with S "@0:D",08,02A7,0304.

- 4) Now for the final change, we will modify program "AB" so that it will look for and load "BA" instead of "9300". Using your monitor, load "AB" and change the values at \$70A8 to FB & \$70AA to 02. This change forces "AB" to look back in our modified autoboot "D" for program "BA". Save this modified "AB" to disk with S "@0:AB",08,61DA,81D5. During actual operation, "AB" is loaded into memory starting at \$A000. You now have a working copy that is error free and has a clean directory.
- 5) I guess a few comments are needed about the encryption mentioned in the "HOW TO" section where we changed a couple bytes to modify the error check routine. You will need a reset switch to see what is happening during the next few steps.
 - (a) Load a monitor that resides at address \$5000 or greater.
 - (b) Now load and run the original D-64 disk.
 - (c) Push the reset switch when the disk drive head starts to bump.
 - (d) Activate your monitor and change address \$0801 from 00 to 43 and \$0802 from 00 to 08. Deactivate your monitor.

- (e) Type LIST and hit RETURN. VOILA!! We have a basic program. And what is this at line 2? Why it's our old friend, the error 23 check routine. Changing the 23 to 00 at this stage of the game does you no good because there's no way to save old "9300" back to the disk. Even if you change the name and make other changes to the directory and "0093" to handle this new name, the next time you run the program it will crash. The reason for this is that the decrypted program that you saved has been decrypted to garbage. Yes, this would have worked if you had found the decryptor and modified or bypassed it.
- (f) One more thing before I fade into the woodwork. Activate your monitor again. Look at addresses \$08B5 & \$08B6, and you will see 32 & 33. This is our error 23 check. Now load "BA" and look at these same addresses. You will see old 84 & 85 that were once a pair of 86's. Bye, for now.
-

The following was submitted by a newsletter subscriber:

The program is Donald Duck's Playground, Copyright 1984 Sierra On-Line Inc.

TYPE OF PROTECTION: This program checks for an error #23. The main program also consists of user files.

HOW TO COPY: This program is easily copied with any good copier capable of reproducing error 23's.

HOW TO MAKE A WORKING COPY WITHOUT ANY ERRORS ON THE DISK:

1. Make a copy of the disk without any errors.
2. Load a track and sector editor and read in sector 17/6.
3. Change bytes number 86 & 93 to \$30 (these represent two comparisons for error #23).
4. Save this revised sector back to the copy disk.

Although this procedure will give you a working copy, you still have the annoying presence of user files which prevents you from saving other programs on the same disk. Unfortunately, resetting your computer after the game is loaded clears important data. The solution lies in the file called "BANNER" which has a curious hidden feature. Do the following:

1. Load "BANNER" from the modified copy.
2. Type Poke 10698,32 (This changes the starting address of the main program from \$3000 to \$2000).

3. Type SYS 8192 (This activates the "BANNER" file).
4. After completion of the load, the program will save itself!!
5. After completion of the save, load your track & sector editor to change the name of the NEW file from \$0D (one hex character) to whatever you like.

Amazingly enough, the original program contains the code to save itself out as a single file! Just goes to show that sometimes it pays to look a little farther after you've done the simplest break. You now have a 178 block file of the original with a starting address of \$3000. You may want to write a boot program to load the file and start it with SYS 12288 (= \$3000).

The program is EASY SCRIPT, COPYRIGHT 1982 BY PRECISION SOFTWARE LTD

TYPE OF PROTECTION: This program will check TRACK 1/SECTOR 0 and TRACK 35/SECTOR 1 for an error 22.

HOW TO MAKE COPY: Copy the program with any good copy program and place an error 22 on track 1/sector 0 and track 35/sector 1, or use a copy program that will place the error for you.

HOW TO MAKE A WORKING COPY OF THE PROGRAM WITHOUT ERRORS ON THE DISK:

There are at least three versions of this program. Two of the versions were submitted by our users. Since we were so close to our printing deadline, we were unable to get the information necessary to assist you in determining the version you have. We apologize for the inconvenience, but can only suggest that you try each technique in the hope that we have covered the one you own. Although we will present the technique for the three versions we have encountered here, you may have another. If what we suggest does not work for your version, examine the code in the areas indicated, and try a new variation.

VERSION 1 (APPEARS TO BE THE LATEST VERSION)

- 1). An expander board is required for EASY SCRIPT (R). A Cardco 5, or a Cartridge Cracker (R) will work. Insert the expander board into the computer, and turn on the switch that activates the EXROM line. This is switch 2 on the OC board, and is the upper-right switch on the Cardco 5.
- 2). Load and execute HI-MON, with SYS 49152. Provide a clean work space with F 8000 BFFF 00. After clearing the work area, execute a reset with G FCE2. The computer should say 30719 Bytes Free.
- 3). Load the original EASY SCRIPT (R) disk with LOAD ":",8,1. When the program is finished loading, the computer will reset.

- 4). Insert a formatted blank disk into the drive. Re-enter HI-MON with SYS 49152. With the M command at \$0001, change the 37 to a 36. This will flip-out BASIC ROM.

DISK VERSION ONLY

- 5). Using the M command, make the following changes to the code:

M 8000 4B - This replaces the 55, which is inserted on the reset. When the EXROM line is grounded during reset, a \$55 will automatically be placed at \$8000. More on this later.
M 82EA EA EA - This change will NOP a branch.
M ABC9 02
M AC5A 02
M B045 A9 23 8D 03 02 A9 4A 85 - This code loads values into areas where the program expects to find them after checking the DOS protection.
M B04D 29 85 3E 60 - This is the rest of the above code, and a return from subroutine (RTS).

CARTRIDGE VERSION ONLY:

- 6). THIS STEP IS ONLY FOR THOSE WHO WISH TO MAKE A CARTRIDGE VERSION OF EASY SCRIPT (R). Make the following changes to the code:

M 8000 4B - See disk version
M 800D FD - Used to switch in the BASIC ROM for cartridge.
M 801F E7 - Switches in the BASIC ROM
M 80B8 FF CF - Transfers the BEEP sound to \$CFFF. The disk version stores the beep within itself. Since a cartridge cannot write to itself, you must change the storage location.
M 82EA EA EA - See disk version.
M AA7F FF CF - BEEP
M AAA2 FF CF - BEEP
M ABC9 02
M AC5A 02
M B045 A9 23 8D 03 02 A9 4A 85 - See disk version.
M B04D 29 85 3E 60 - See disk version.

DISK AND CARTRIDGE VERSION:

- 7). Save the altered code from 8000 to B66A with S "EASY SCRIPT",08,8000,B66A. To load the program, type LOAD "EASY SCRIPT",8,1 and SYS64738 to activate.

CARTRIDGE VERSION:

- 8). If you altered the code so it may be placed on a cartridge, you must "burn" the code to two 2764 eproms. The first eprom should contain the code from 8000 to 9FFF, and the second eprom should contain the code from A000 to B66A.

9). YOU'RE DONE.

VERSION 2

In the first version, we defeated the auto-start through the use of the cartridge exrom switch. In this version, we will change one of the bytes of the CEM80 to accomplish the same thing.

- 1). Make a copy of the original disk without errors. If you do not have utility program that will change an E at 18/0 to an A, you will have to copy around this block with BACK-UP 228. Once you have all but 18/0, load the original at 18/0 through your track and sector editor and change the E to an A. Use your write feature to write this track and block to your copy disk. With this accomplished, go on to step 2.
- 2). With your track and sector editor, go to track 35/sector 0 and change byte 6 (don't forget to start counting with 0) from a B6 to a B0 (HEX MODE). This is the code that will be modified to a CEM80. Some track and sector editors will show this code as an inverse 650.
- 3). Load and execute HI-MON, with SYS49152. Clean-up the work space, with F 8000 BFFF 00. Execute a reset with G FCE2.
- 4). Load the program as usual, with LOAD":*",8,1. Once in memory, the computer will reset to basic.
- 5). Re-execute HI-MON, with SYS49152. Using the M command, go to \$0001 and flip-out the BASIC interpreter by changing \$0001 from a 37 to a 36. Using the M command, make the following changes to the code:

M 8004 C3 - Restores the CEM80 (CEM80)
M B047 4C B0 92
M B092 A9 4A - Checking for the "E" at 18/0.
M B096 4C C9 B0
M B0C9 A9 32 - Stores ASCII 2 error code.
M B0DC 4C 08 B1
M B110 60
M B0CE THROUGH B0D4 EA - Storing EA's in these memory locations removes the error-checking.
- 6). Insert a formatted disk and save the code with, S "EASY SCRIPT",08,8000,B66A. You may now load the program with, LOAD "EASY SCRIPT",8,1 and activate with SYS64738.

7). YOU'RE DONE.

VERSION 3

If you have the third version of the program, follow the steps outlined in VERSION 2 to defeat the auto-start and make the following changes to the code:

- 1). Make a copy of the original disk without errors.
- 2). With your track and sector editor, go to track 36/sector 0 and change byte 06 from a B6 to a B0 (HEX MODE). Again, this is the code that will be modified to a CBM80. You will recognize the code on the disk by the inverse 650.
- 3). Load and execute HI-MON, with SYS49152. Clean-up the work space, with F 8000 BFFF 00. Execute a reset with G FCE2.
- 4). Load the program as usual, with LOAD":*",8,1. Once in memory, the computer will reset to basic.
- 5). Re-execute HI-MON, with SYS49152. Using the M command, go to \$0001 and flip-out the BASIC interpreter by changing \$0001 from a 37 to a 36. Using the M command, make the following changes to the code:


```

M 8004 C3 - Restores the auto-start (CBM80)
M B047 4C 92 B0
M B092 A9 4A
M B096 4C C9 B0
M B0C9 A9 32
M B0CE through B0DB EA - Removes error checking.
M B0DC 60 - Return from subroutine (RTS).
      
```
- 6). Save the code with, S "EASY SCRIPT",08,8000,B66A. You may load the program with, LOAD "EASY SCRIPT",8,1 and activate with SYS 64738.
- 7). YOU'RE DONE.

The program is **ECONOMICS**, Copyright 1983, Micro-Ed, Inc.

TYPE OF PROTECTION: The program disk is littered with errors. The program will check for an error and run if present.

HOW TO COPY: Use a copy program that will place the errors for you.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK.

As with the previous two programs, we are faced with hidden lines. The unprotection process for this program is an easy one, so let's get to it.

1. LOAD and RUN the program in the usual manner. When the MENU appears, choose the first program and allow it to run normally. Once in memory, press the RUN/STOP and RESTORE keys.
2. You may now save the unprotected program to a formatted disk with SAVE "EC1 DOLLAR & CHA",8

3. Repeat STEPS 1 and 2 for the other five programs on the disk.
 4. With the six programs saved to disk, we will now alter the menu program so that it may be used without the protection scheme.
 5. LOAD "MICRO-READY",8.
 6. The lines from 10 through 59090 contain the error protection. Move to the bottom of the screen so that we can delete these lines. Type the following:

10 (RETURN)
80 (RETURN)
90 (RETURN)
9999 (RETURN)
59002 (RETURN)
59010 (RETURN)
59020 (RETURN)
59030 (RETURN)
59050 (RETURN)
59060 (RETURN)
59090 (RETURN)
 7. The menu is now unprotected and may be saved to your formatted disk under any title you wish.
 8. YOU'RE DONE!
-

The program is F-15 STRIKE EAGLE, Copyright 1984 by MicroProse Software.

TYPE OF PROTECTION: The disk has duplicate sectors 1-9 on tracks 1-3, and error 29's on track 35. The protection routine uses encryption and undocumented opcodes.

HOW TO COPY: Some of the newer, better nybble copiers can copy this program. The duplicate sectors make it difficult to copy.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS:

- 1) Copy the original disk with any copy program which does not put errors on the backup copy. You may even file copy the disk if you prefer.
- 3) Load in HIMON. Using the monitor, load in the file containing the protection routine: L "TITLE.BA",08.
- 4) Change memory locations \$0EF0-0EF3 from \$B2 9D F1 to \$14 14 14 (encrypted NOP's).

- 5) Change locations \$0F5D-0F5F from \$DE 2F FF to \$14 55 B2 (encrypted form of NOP LDA #\$4C).
- 6) Replace the changed file with S "@0:TITLE.BA",08,0801,0FFB.
- 7) That's it - you're done!

This program is interesting because it uses a combination of protection methods. The file TITLE.BA consists of two parts, a BASIC program to print the title screen and an ML section to check the disk protection. The BASIC program calls the ML routine with SYS 3416 at line 301. Line 301 is hidden from being LIST'ed, using delete characters (see the PPM Vol I for a discussion of this technique). To make the line listable, change the deletes (\$14) to spaces (\$20) with an ML monitor.

The ML routine makes extensive use of undocumented opcodes. They are used immediately upon entry at \$0D58 (= 3416 decimal) to alter the routine so that it jumps to \$0E28 next. This begins a process which decrypts some of the remaining code and executes it. As this code executes, it decrypts several disk commands one-by-one and sends them to the drive. First, the routine checks track 35, sector 3 for an error 29. This check is disabled by the patch made in step 5 above. Then it loads track 3, sector 3 into the drive and reads the first byte. This byte is supposed to be a \$4C, so the patch in step 6 simply replaces the call to CHRIN (Kernal routine at \$FFCF to get a character from the drive buffer into the accumulator) with NOP LDA #\$4C. The \$4C is then stored in the computer's memory at \$DFFF, in the RAM underneath the I/O devices. Later on it is checked and the program crashes if the value isn't correct.

The decryption process used in this routine is fairly simple. The encrypted section of memory runs from \$0E2D to \$0FFB. If you want to examine the protection method, you'll have to decrypt this section first. Note that the routine still contains some undocumented opcodes, so it will not disassemble completely with a standard monitor (see the PPM Vol II for a table of undocumented opcodes). To decrypt a byte, you simply clear the carry flag, add 1 to the byte and exclusive-or (EOR) the result with the value \$FF. Clearing the carry is necessary each time since the addition instruction automatically adds in the carry flag. The following piece of code will decrypt the first 256-byte page of the code. Modifying it so that it will decrypt the entire section of code is left as an exercise for the reader.

```

C000 LDY #$00
C002 LDA $0E2D,Y      $0E2D = starting address
C005 CLC              Clear carry flag
C006 ADC #$01         Add 1 to accumulator
C008 EOR #$FF         Exclusive-or
C00A STA $0E2D,Y      Replace decrypted byte
C00D INY
C00E BNE $C002
C010 BRK

```

By the way, this decryption method is also self-reversing, i.e. the exact same routine can be used for both encryption and decryption. Thus you can decrypt the code, modify it as you wish, and then encrypt it again using the decryption routine. Many of the common, simple encryption methods using EOR have this property. See PPM Vol II for an explanation of encryption and decryption techniques.

F-15 STRIKE EAGLE - UPDATE

Thanks to Joe Aufiero of Scotia, N.Y. for the following information.

There are apparently different versions of F-15 on the market. If the modifications given last month don't work for you, try this:

1. Load in "TITLE.BA" with HIMON. Line 301 contains a 'hidden' SYS 3416, which must be disabled by changing the \$9E (SYS) to an \$8F (REM). This byte is located at \$0BEF in the version that we have, but yours may be different. Save the file back out with S "@0:TITLE.BA",08,0801,OFFB
2. Load in "F15.VID". Change bytes \$7196-9D from \$FF DF DD 90 71 F0 52 20 to EA EA EA EA EA 4C EF 71. Again, these bytes may be at a different location in your version, so hunt for them. Save the file back out with S "@0:F15.VID",08,5100,7E00.
3. That's it.

The program is **THE FACTORY**, Copyright 1983, Sunburst Communications.

TYPE OF PROTECTION: This program will read TRACK 18 SECTOR 2 for the ASCII values of SB. If present the program will run properly. This program appears to be totally unprotected in its original form, but the ASCII SB is the program's protection 'key'.

HOW TO COPY: Copy with any good copy program that will copy the entire disk. Alternatively, a file copy of the disk may be performed and the SB bytes placed on track 18 sector 2 at bytes 149 and 150 respectively.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

As you can see from above explanation the program does not require any errors on the copy disk. As like all other protected programs there is still a 'KEY' (in this case the SB) required on the disk that is necessary for the proper operation of the program. The 'KEY' may be anything on the disk. Bad blocks, modified sectors, extra tracks, track arcing, nybble counting, density changes, synchronized tracks and even just a couple of ASCII bytes on a sector are all forms of protection 'KEY's. The 'KEY' is the required element on a disk that the program checks for. Without the proper key the program will crash; with the proper key the program will execute normally. We are seeing a great deal of

programs where the disk appears normal in most respects, yet there is still something on the disk that is required for the proper operation of the program. This type of protection may be very sophisticated or very easy. Just don't assume that the programmer is using some super new scheme when you encounter this type of protection. Many times, if the program is just using the more basic types of schemes we find a B-P (BUFFER-POINTER) command used in combination with the protection scheme.

In this program, the checking is done in the program called "SBSOGOB". This is a basic program that loads in a machine language program. LOAD "SBSOGOB" and list LINE 600. This is where you will find the B-P. It is calling for a check of 18/2 and looking for an SB at byte 149 and 150. The program will act on the result of that check in LINE 15050. If the result is not equal to "SB" the program will be sent to a SYS that will crash the program.

Normally we would just change the < to an =, but we must take care to keep the line length the same so that the machine language program that follows the BASIC resides in the proper location. An = will leave us one byte short in this line, causing the entire machine language section to be one byte too low.

1. Copy the disk with any good copy program that will copy the files only. Alternatively, copy the disk with a full disk copier and with a track and sector editor at TRACK 18 SECTOR 2, locate the SB. Change to HEX mode and change the 5342 (SB) to 0101.
2. LOAD "SBSOGOB",8
3. Now LIST 15050. Our task is to alter the code while keeping the line length the same. Many protection schemes written in BASIC will check the length of the BASIC program to detect if the program was tampered with. It is important to maintain the same length of the program when modifying some BASIC protection schemes. We will accomplish this task by inserting a space in place of the < and changing the > to an =. The line should now read:

```
FORDL=1TO4000:NEXTDL:IFA$="SB"THENSYSAA
```

4. Save the altered code with SAVE"@0:SBSOGOB",8
5. YOU'RE DONE!

Keep this technique in mind when unprotecting BASIC programs. You may even find programs that will execute a checksum to determine if the original code has been altered.

This one is for 128 owners only!

The program is **FLEET SYSTEM 3**, Copyright 1984 by Visiontronics Ltd. and Professional Software, Inc.

TYPE OF PROTECTION: The program will check the disk for the proper coding. If not found, the program will RESET and try to reboot.

HOW TO COPY: The copy programs we have tried will not copy this program.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

This program does not check for errors once it is in memory. We will make a working version of the program by saving out the memory used by the program and making a boot.

1. In order to have the proper printer codes, you must first run the SETUP file included on the original disk. Follow the instructions in your FLEET SYSTEM 3 manual to install the printer codes. You should check that the program prints correctly before proceeding.
2. We'll start by creating a boot to load in the two sections of code we'll save and RUN them. Enter the following:

```
10 PRINT CHR$(14) :REM LOWER CASE
20 IF L=0 THEN L=1 :LOAD"STACK",8,1
30 LOAD"FLEETMAIN",8
```

Now save the boot to the COPY disk with SAVE "FLEETBOOT",8.

3. Power-down your C128. Enter the machine language monitor with F8 (SHIFT F7). Clean up the work space with F 0100 01C0 99 and F 1C01 9FFF 99. Exit the monitor with X followed by a RETURN.
4. Load the program from the original disk with LOAD"FS",8 and RUN.
5. While the main body of the program is loading, watch the disk drive light. As soon as the light begins to flash, RESET to the monitor by holding down the RUN/STOP key and then pressing the built-in RESET button. You will see the usual monitor display.
6. The program code is stored in BANK 0 from \$1C01 through \$8A1F. Type M 1C01. you'll see a SYS 7181 statement. Using M 89C0 reveals that the program code ends at \$8A1F, which is where our \$99's begin.
7. Save out the main body of code to the COPY disk with S "FLEETMAIN",08,1C01,8A20. Remember, we always use the end address +1.
8. We will now save out a section of the stack, which contains the printer codes. S"STACK",08,0100,019F should do the trick, but you should display the first part of the stack (M 0100 01C0) and look for where the \$99's end to be sure.
9. YOU'RE DONE. To use, type LOAD"FLEETBOOT",8 followed by RUN.

Before we close the book on this one, just one word of caution. If you RESET too late, the THESAURUS will not work properly. Check this feature after running the backup you just created. If you have a problem, just begin again and RESET as soon as the disk drive begins to flash.

THE FOLLOWING PROGRAM USES A SPECIAL TECHNIQUE THAT WE FEEL OUR READERS WILL BE ESPECIALLY INTERESTED IN LEARNING!! We feel most (if not all) of our readers can learn from this protection method.

The program is FLIGHT SIMULATOR II (C) 1984 by Bruce Artwick, produced by SubLOGIC CORP.

TYPE OF PROTECTION: This program uses an error 21 on all of track 3. As the program boots up it checks track 3, sector 00 for an error 21. It also checks tracks 1, 2, 4, 5, 6, 7, 8 and 9 to insure that there are no errors on these tracks. While the check of tracks 1-9 does not take very long, it is unnecessary and adds to the extremely long load time.

WARNING: This type of protection can be hazardous to your drive's health.

HOW TO COPY: Any good copy program is capable of copying this program. All that has to be done is to be sure that there is an error 21 on track 3, sector 0. ALTERNATELY: you can use SNAPSHOT 64 (tm) to make an archival copy (if your program doesn't have sound, simply restart the engine). ISEPIC (tm) does not appear to be able to do this program (and similar programs) because it apparently doesn't get the raster interrupt value. We haven't seen CAPTURE yet, so we aren't able to test its effectiveness.

HOW TO MAKE A WORKING COPY WITHOUT ANY ERRORS ON THE DISK:

1. Make a copy of the disk without any errors on the disk.
2. Use a track and sector editor to edit track 1, sector 5 of the copy disk. Change bytes \$D8, \$D9 and \$DA as follows:

LOCATION	ORIGINAL	MODIFIED
\$D8	20	8D
\$D9	48	26
\$DA	78	73

3. YOU'RE DONE!!

Before we cover the specifics of how this routine works let's just cover a few basics. The program uses an autoboot that loads in a number of blocks from track 1 and then executes that code. These blocks from track 1 are loaded in sequentially from sector 0 (stored at \$7300 in the computer) to sector 8 (ending at \$7AFF in the computer). The data stored on the disk is not encrypted or encoded in any way and there is a direct one to one relationship between the bytes on the disk and the bytes in the computer. The code that resides from \$7300

to \$7AFF actually does the protection check. It opens files, opens channels, sets file names, prints the UI to the disk drive, reads the error channel, closes the channels, closes the files and it does this all without performing ANY KERNAL calls. In fact, both KERNAL and BASIC ROMs are flipped out by this program

'How is this possible?', you ask??? - Glad you asked - we're gonna tell ya. Remember back about two months ago (AUG. 85 issue), we wrote a couple of pages in the NEWSLETTER about ALTERNATE KERNAL DISK ROUTINES. Well we didn't just write this for the sake of filling up the newsletter, we wrote about it because IT IS IMPORTANT. Those of you that didn't read it, SHOULD!! Those of you did read it are fortunate enough to have a couple of months to mull the information over in your mind. We hoped that the article stimulated some thought about alternate KERNAL routines, because this program uses them. These routines can be quite hard to follow so we will try to give you some help.

First, we want to give you just a brief explanation of what you did when you changed the three bytes on the disk. The code that you changed resides in the computer at \$78D8 and is a JSR \$7848 (20 48 78). This subroutine (at 7848) is the code that actually checks tracks 1 thru 9 for the proper error condition. If the disk is in its original form (error on track 3) this code will end up placing a \$00 at location \$7326. If the disk's condition is not in its original form, a different value will be placed at \$7326 by the subroutine. Later on in the program the value at \$7326 will be checked. If it's 00 the program will run. If it's not 00 the program will crash (no pun intended). All we did was to replace the JSR \$7848 with the code: STA \$7326 (8D 26 73). At this point in the program the ACCUMULATOR contains the value of \$00 (it was just loaded with \$00 a few instructions earlier). So all that is necessary is to bypass the protection entirely and simply store the appropriate value where needed (by changing a few bytes as outlined above). By the way, the program actually checks location \$7326 for the proper value with a routine located under the KERNAL (\$E56F to \$E58E). This part of the program is loaded after the protection check. Now on to the alternate KERNAL routines.

This program is a real learning experience when it comes to the alternate KERNAL routines. As a matter of fact the code from \$7300 to \$7AFF does not even use a single part of the ROM chips for its communication with the disk drive. This code contains all the routines necessary to communicate over the serial bus. The code appears to be a direct byte-by-byte re-creation of the necessary KERNAL calls. For instance: the code at \$7752 to \$7763 is performs the identical function as the KERNAL's CIOU routine (located at \$EDDD). The CIOU routine sends a byte over the serial bus with full buffering. The KERNAL CIOU routine calls a subroutine located at \$ED40 which actually outputs the byte. Similarly, the code from \$76B7 to \$7721 does the identical function as the routine in the KERNAL at \$ED40. While the actual code used by FLIGHT SIMULATOR is not totally identical byte-for-byte, the ALGORITHM is identical. In other words, while the routines are not exactly the same, the exact same functions do get accomplished. If you wish to take a little time and examine the code further you will find many other byte for byte occurrences.

When you examine the code used in programs that use alternate KERNAL routines the best tool that you can use is a ML monitor. It's possible to compare different areas of memory with most ML monitors using the 'C' or the 'H' commands. In a program such as this, that totally relocates the KERNAL routines, you will find that many ABSOLUTE references will be changed. For instance, in the KERNAL routines the JSR \$ED40 (at \$EDE7) was changed to the JSR \$76B7 (at 775C) in the alternate KERNAL routine used here. At the same time the BIT \$94 (at \$EDDD) was changed to BIT \$BA (at 7752) in the alternate KERNAL routine. You'll find that all the branches (relative references) will remain the same. In this program you could simply use the 'H' command to see if the routines actually came from the KERNAL. Don't hunt for any code that contains absolute references, use only those locations that contain relative references or implied addressing (branches, PHA, PLA, SEI, PHP, PLP, TXS, etc.). For instance, we've told you that the range of code from 7752 to 7763 performs the exact same function as the KERNAL routine called CIOUT (\$EDDD - EDEE). In this range of code you should hunt for the three bytes of code that contains the following instructions:

```
7759 D0 05      BNE $7760      (relative branch)
775B 48         PHA           (implied addressing)
```

In order to hunt for this code in the KERNAL use the following command:
H E000 FFFF D0 05 48 (RETURN)

The location returned by the hunt will be \$EDE4. An examination of the code adjacent to location \$EDE4 will reveal the similarities that we mentioned. Further study will show that these routine actually perform the exact same job.

This program posed an especially interesting and enlightening look at the actual use of alternate KERNAL routines. In future programs we will be seeing more of the routines appear. Be sure to take a little time to examine, understand and learn from this example. Remember, when we present a new concept in the newsletter it is because of its growing viability as a protection scheme.

The program is GARFIELD, (C) 1986 by Developmental Learning Materials.

TYPE OF PROTECTION: Track 1, sector 1 is checked for an error 29.

HOW TO COPY: Any good nibbler will copy this disk.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

1. Format a disk for the backup (any name/ID will do). Using a file copier, copy all the files from the original disk to the backup disk.
2. Load in the file called "!" using LOAD"!",8. List line 110 (the program is in BASIC). Change the number 29 in the IF statement to 00.

3. Save the modified file back to the BACKUP disk with SAVE "@0:!",8. If you don't want to use SAVE @, just scratch the file first using OPEN15,8,15,"S0:!" and then save the new version using SAVE "!",8.

4. The disk will work as-is, but one file name contains DELETE characters (CHR\$(20) or \$14 hex). To fix this, rename the file using:

```
OPEN15,8,15:PRINT#15,"R0:GARXXX=0:GAR"+CHR$(20)+CHR$(20)+CHR$(20)
```

The "XXX" we used in the new name could be replaced by almost any three characters. The "!" program loads this file using "GAR???", which just specifies any six-letter name starting with "GAR".

5. That's it - you're done!

This protection scheme is a real anachronism! I thought the days of a BASIC error check were long gone, but here's one in a program copyrighted in 1986. Not only that, but the programmer has thoughtfully labeled it for us with a REM >>> CHECK FOR ERRORS <<< statement! I was sure this was just to make you overconfident, to hide the "real" check farther in the program. However, no amount of play-testing or code-tracing could turn up another check. Everything works as it should.

The first file on the disk, "LOADER", is a BASIC program that just loads in the file called "GARFIELD" at \$00C6-02E0. This file uses the "dynamic keyboard" technique to load the next file: a command is put into the keyboard queue at \$0277 and the number of characters in the command is put at \$00C6. The command that gets put in the queue is SYS680:RUN. The SYS 680 executes a machine language routine at \$02A8 (which equals 680 decimal). The ML routine just loads in the BASIC program "!" and returns. The "!" file is executed by the RUN command still in the queue.

In line 30, the "!" program disables the disk drive's head "bump" (see the Oct. 84 Newsletter). This is VERY strange since an error 29 is the one error that does NOT cause the head to "bump"! This was another thing that lead me to think there might be a second protection check (sector 1/16 on my disk did show an error 20 with ONE error scan utility - the rest of track 1 is all error 29's). The error 29 at sector 1/1 is checked using a "U1" command. If the error is NOT found, the program cold-starts BASIC with a SYS 58260 (\$E394). If the error IS found, all the program does is load a file using the name "GAR???" (see above) and RUN it. In fact, if you just load "GAR???" and RUN it, the program seems to work. Just to be on the safe side, though, we've left the boot process as close to the original as possible. Our change in step 2 just reverses the results of the error check after the "U1" command.

The program is GEOS Version 1.0, copyright 1985 by Berkeley Softworks.

TYPE OF PROTECTION: Track 36 is checked for three particular GCR bytes.

HOW TO COPY: No nibbler we tested would make a duplicate of this disk directly. However, the latest Fast Hack'em parameter disk has a parameter to reproduce the protection on track 36. This means a backup copy made with Fast Hack'em will still be protected.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

1. Make a copy of the disk with no errors on it.
2. Load in a track/sector editor. Insert the COPY disk in the drive.
3. Read in track 1, sector 11. Change the byte at position \$F0 (240) from \$0D (13) to \$52 (82). Save the sector back out to the COPY disk.
4. That's it - you're done. The disk can be copied with any whole disk copier, but NOT with a file copier. It must still be loaded from a 1541, 1571 or "1541-clone" drive (sorry, no MSD's).

The first file loaded is called "GEOS". It loads into the stack area, at \$0110-0205. Like most autoboots which use the stack method, it fills the last part of the stack with \$02 bytes. This causes a jump to \$0203 when the LOAD routine finishes. The code at \$0203 then jumps to \$0123, which starts the boot process. The code at \$0123 simply sets screen colors and loads in the next file, "GEOS BOOT", at \$1000-1463. Then it jumps to \$1000, which immediately jumps to \$106C. The code at \$106C decrypts the rest of the file, starting at \$1086, by EOR'ing (Exclusive-OR'ing) each byte with the value \$27. To see the decrypted code, load in "GEOS BOOT" with a monitor. Change the byte at \$1086 from \$5F to \$27. Now execute the decryption routine using G 106C. In a flash you'll be returned to the monitor. What did we do? Disassemble the code starting at \$1086. There you'll see a BRK instruction (\$00 byte). This is what popped us back into the monitor after the code was decrypted. The \$27 byte we put at \$1086 was simply the encrypted form of \$00 (the \$5F that was there originally was an encrypted \$78, SEI). See PPM Vol. II for chapters on encryption and autoboots.

The decrypted code prints a screen message, using routines at \$11C5, \$11E4 and \$11FD. Then it sets things up for a fast load routine. The fast loader requires a custom DOS routine in the drive, which also includes the protection check code. At \$10E8, the DOS routine is sent to the drive, using routines at \$1227 and \$1246 to perform M-W (memory-write) commands. After the routine is sent, it is executed via an M-E (memory-execute) command. Let's leave the fastload/protection routine alone for a minute and see what happens next in the computer. To understand it we have to detour into a special feature of GEOS.

GEOS has its own custom file type called Variable Length Indexed Relative (VLIR). They show up as USR type files in the directory. Normal files (including USR and REL types) are composed of a chain of sectors linked together, with the first link stored in the directory entry. A directory entry has many unused bytes in it, however, and GEOS uses the extra bytes to store several more starting links. This allows VLIR files to be composed of several different parts, each one

an independent chain of sectors (this is why a normal file copier won't copy them).

The file "GEOS KERNAL" is loaded next via the fast load routine. It's a VLIR file with three parts, loaded at \$9000-9FFF, \$C000-CFFF and \$D080-FFF7 respectively. At \$113A the program sets up a pointer (\$04-05) with the load address of the first section (\$9000). Then it calls a subroutine at \$1181-B4, which loads in one entire linked section. As the section is loaded, the pointer at \$04-05 is incremented. However, if the fast loader in the drive doesn't find the correct protection on the disk, it won't transfer the first section of code to the computer. The pointer at \$04-05 in the computer will NOT be incremented since nothing was loaded. This pointer is checked after the load. If it's still \$9000, the computer knows the protection check failed and it RESETs itself at \$114B. We can't just disable this check because the first section of code still won't be loaded. We'll have to get into the DOS routine to fix that. On the other hand, if the protection check succeeds, the program goes on to \$114E and loads in the other two sections. Then it starts the main program with JMP \$C003.

Now let's look at the DOS routine. It resides at \$1263-1463 in the computer, and is transferred to \$0300-0500 in the drive. To examine it more conveniently, transfer it up in memory using T 1263 1463 2300. The routine is normally executed at \$0375 in the drive, which will be at \$2375 in the computer now (you'll have to make this adjustment yourself from now on). After a little setup, the main routine at \$042A-0500 is called and immediately begins checking the protection. The main routine calls a subroutine at \$03AB-0408, which steps the head to the track indicated in the accumulator and sets the proper density for the track from a table in the ROM. In this case, however, it is moving to track 36 where the protection values are written. Track 36 is not normally used and does not have an entry in the density table, so the subroutine uses the density for ZONE>1 (tracks 1-17). Then it returns to the main routine to start the protection check.

The protection consists of three consecutive GCR bytes, \$62>73>77, written on track 36 at zone 1 density without any SYNC marks. The lack of SYNC marks makes it very difficult to copy the track, as well as making it impossible to tell where the first byte begins. To get around this, the group of 3 bytes is written on the track at least 8 times, with one extra bit« written between each group of bytes. This way, no matter where the head comes onto track 36, by taking the bits 8 at a time it will eventually come to a copy of the 3 bytes that it can recognize. It counts the bytes as it reads, and the protection fails if it reads 32768 bytes without finding the 3 it's looking for. That's about 5 times around the track, for good measure. The protection also fails if it encounters a SYNC mark as it reads.

At \$042A it sets up the byte counter and puts the head in read mode (which it's already in). At \$043E it increments the counter and checks it. At \$0448 it checks for a SYNC mark. Then at \$044D it starts looking for the \$62 73 77 pattern. Each time it fails, it branches back to increment the counter and check for a SYNC mark. If the counter runs out or a SYNC is found, the protection has failed. In that case, it branches to \$046D to signal the computer that the first part of the file is finished loading, although nothing has actually been loaded.

On the other hand, if the protection is found, it branches to \$0475 to load the first part of the file properly. Disabling the protection check is easy - we'll just skip that section of code. This requires changing the statement at \$0384 from JSR \$042A to JSR \$0475. The change in step three replaces the encrypted 2A with an encrypted 75. Voila!

The program is **GHOST CHASER** by Frank Cohen. Copyright 1984 by Fanda.

TYPE OF PROTECTION: The disk has error 21's on all of tracks 2-5. Block 14/10 has an error 23.

HOW TO COPY: Use any good fast copy which copies bad block errors.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS:

1. File copy the original disk with a suitable file copy program.
2. Load and run HIMON or other monitor above \$4000.
3. Load in the file "C" from the copy disk: L "C",08
4. Make the following changes:

LOC	FROM	TO
\$4097	\$57	\$77
41B8	A0	A7
4228	57	77
4232	57	77

5. Replace the altered file: S "@0:C",08,0900,42B0
6. That's it - you're done!

This program uses a variety of protection methods. The disk name ends in a \$00 byte, which prevents the directory from being LIST'ed from BASIC. Several of the files are encrypted, including the protection routine. After decrypting itself, the routine checks for one error 21 somewhere on tracks 2-5 (any sector). It uses the jiffy clock to randomly select both the track and sector within this range. If an error 21 is found in the block chosen, the three remaining files on the disk are loaded into memory. Next, the routine checks block 18/0. This block contains the disk name and BAM, which will be changed if you add files or rename the archival disk. The bytes of the block are ADD'ed together and the one-byte result is compared with a prestored value. If they are equal, the routine checks block 14/10 for an error 23. If the error is found, all is fine. The routine decrypts two more pieces of code and then the program begins its normal operation.

As you can tell, the protection routine uses a good assortment of techniques. Unfortunately, it hammers the head two separate times in the process. Fortunately, we can do something about that. We discovered the protection methods by simply tracing the program right from the start. The average reader should find it an interesting and informative exercise. Other readers may find it plain good practice. If you can find this program for sale, you should buy it and try to trace the protection without looking at the rest of this article. The previous paragraph should give you a good headstart.

Let's see how you did. The boot (Ghost Chaser) loads in at \$02A7-0333, replacing part of the BASIC vector table that starts at \$0300. As usual with boots at \$02A7, this alters the BASIC warm-start vector stored at \$0302-0303. Instead of the normal value, \$02A7 is stored there. Recall that vectors are stored in lo byte/hi byte (reverse) order. In particular this means that location \$0302 contains the value \$A7. The value in this location is important because it is used later to decrypt several files. When the boot executes it loads in the files "A", "B" and "C" and then jumps to \$4000 (in file "C"). File "C" contains the entire protection routine plus some other routines. Files "A" and "B" are the color RAM and screen memory for the display shown while the program loads.

The protection routine begins right at \$4000. The first section of code decrypts the remaining section (\$4026-42B0). It does this by using the EOR instruction to exclusive-or each byte with the value in location \$0302 (\$A7). We can force it to decrypt itself for us so we can look at it. Load the file "C" from HIMON, put the value \$A7 at \$0302, put a BRK (\$00) at \$4025, and G 4000. By the way, you may have noticed that the BRK you inserted at \$4025 wiped out a NOP instruction that was there. This NOP was undoubtedly left by the programmer to replace the BRK he/she used when the routine was encrypted. As with all straight EOR encryption, the same procedure can be used to encrypt AND decrypt equally well. The programmer may well have used exactly the same procedure we did!

The decrypted code begins by initializing the drive ("I" command) and opening a random access ("H") channel. Next it sets up the track and sector for a "U1" command. It calls a routine at \$40BE, which uses the jiffy clock to randomly select a track in the range 2-5 and a sector in the range 0-20. The jiffy clock is located at \$A0-A2. It is a THREE-BYTE counter incremented by the normal IRQ routine every 1/60 second. The sector is based on the value of \$A2 (lo byte), and the track is based on \$A1 (med byte). The track and sector bytes are checked for the correct range and then converted into ASCII by a routine at \$40E0. Finally they are stored into the "U1" command at \$4128-29 (track) and \$412C-2D (sector).

After sending the "U1" command, the first digit of the error number is input and CMP'ed to \$32 (bad block errors start with an ASCII "2" = \$32). The first change in step 4 above alters an encrypted BEQ to a equally encrypted BNE in order to reverse the effects of this check. Next, the second error digit is input and CMP'ed to \$32 too. A BOC instruction then checks whether the digit is LESS than \$32 (it's expecting an error 21). We DON'T change this check because it will still work correctly when the error number is 00 (no error, as on our archival disk). Just goes to show you shouldn't automatically change CMP \$32 code until

you've analyzed it to see what it really does. Once it is past the error 21 check, it jumps to \$412F and loads in three files - "D", "E" and "F".

Next it uses a "U1" command to read in block 18/0. It ADC's each byte into a running total at \$41DF and then CMP's the final result to the value \$7A. If they're not equal, it branches (BNE) to a routine that crashes the program. The second change in step 4 above alters the target of the BNE from the crash routine to the very next instruction. Now it doesn't matter what the total is. Either way the routine will just go on to the next instruction. Once past the 18/00 test, the routine then checks block 14/10 for an error 23 using yet another "U1" command. Both digits of the error number are checked to make sure it is exactly an error 23. The last two changes in step 4 turn the BEQ used to test each digit into a BNE. After passing all of these protection checks, the routine decrypts the rest of file "C" (\$0900-3FFF) and all of "D" (\$4800-A1FF). It also re-encrypts the first decryption routine (\$4000-25) to hide it again. This decryption and encryption is done exactly the same way the first decryption was done, by EOR'ing each byte with the contents of \$0302 (i.e. \$A7). For the grand finale it jumps to \$0900 to start the game.

The program is **HEART OF AFRICA**, copyright 1985 by Ozark Softscape. Published by Electronic Arts.

TYPE OF PROTECTION: EA has used a new type of protection on this disk (or a new variation of their old protection).

HOW TO COPY: We could not find a nibble copier or EA backup program on the market today that would copy this program.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

This procedure requires the use of a SNAPSHOT64 cartridge. Before using SNAPSHOT, we must follow a special process to clear room on the backup disk for the SNAPSHOT files. This way the backup will only take one side of a disk for all the files rather than two.

1. Make a backup copy using any copy program that does not write errors.
2. Insert the backup copy disk into drive 8 and enter the following:

 OPEN15,8,15,"S0:*" :CLOSE15 (Scratch all program files)
 OPEN15,8,15,"V0" :CLOSE15 (Unallocates all 664 blocks)
3. Load and execute a track/sector editor. Read in track 18, sector 0 and the following changes: Bytes \$4C-8F: Change to all \$00's

Note that the bytes are numbered in the sector starting at \$00. In step 2 we deallocated all of the blocks (except for the directory on track 18). This step re-allocates tracks 19-35 but leaves tracks 1-17 available for the

SNAPSHOT files to use. Tracks 1-17 of the original disk contain the main program, which we will SNAPSHOT. Tracks 19-35 contain information that is used throughout the rest of the game.

4. Now we will SNAPSHOT the main program. Power off the computer and insert the SNAPSHOT cartridge.
5. Turn the computer back on. When it is ready, depress the SNAPSHOT button. The screen will blank. Press F3 to clear the memory. This should be followed by a SYS64738 to reset computer (the program wouldn't load correctly for us unless we did this).
6. Insert the ORIGINAL program into drive 8 and load it as usual.
7. When the menu screen appears (next screen after the snake), press the SNAPSHOT button. Turn the drive off and back on (or just RESET the drive) and insert your BACKUP disk.
8. Press F1 to begin the SNAPSHOT process. Answer the prompts for file names as usual.
9. When SNAPSHOT is done, so are you!!!

This procedure must be followed in order to merge the SNAPSHOT files with the data contained on the ECA disk. Although SNAPSHOT will save the main program, the program still accesses the disk as the game continues. The backup disk can now be copied with any copy program.

The program is THE HEIST, COPYRIGHT 1984 MIKE LIVESAY COMPUTER GAMES INC.

TYPE OF PROTECTION: This program checks track 18 sector 18 for an error 23 CHECKSUM ERROR IN DATA. If this error is present, the program will run properly.

HOW TO COPY: Copy the disk with any good copy program and place the error 23 on the disk at 18/18. You may also use a nibble copy program and let it put the errors on for you.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK.

The technique used on this program is similar to the techniques described on pages 50-60 of the Program Protection Manual.

- 1) File copy the HEIST and HEIST.OBJ files to a disk without errors.
- 2) LOAD and execute HIMON (a ML monitor residing at 49152-\$C000). Fill memory from 0800 to 9FFF with 00's. This will make it easy to locate the beginning and ending addresses of the program. Finding the starting address is explained in the P.P.M. on page 35. Load "HEIST.OBJ". The code is located

in memory from \$0800 to \$7FFF. Using the 'I' command of your ML monitor, you can easily find the ending address. Using the 'H' (hunt) command, you will find the "U1" located at \$23BA (U1:5,0,18,18). If you scroll up to \$237E, you will find the CMP #\$32. Also, located at \$2385, you will find the CMP #\$33. Change any errors on the disk. CAUTION: You must change both locations. This program checks for a specific error, not just any error.

- 3) Save the code out to the copy disk. S "@0:HEIST.OBJ",08,0800,7FFF
- 4) YOU'RE DONE.

The program is HESMON 64 (TM) COPYRIGHT 1982 BY HUMAN ENGINEERED SOFTWARE. An excellent ML monitor. If you have not purchased this program I would highly recommend you do. It is very good.

TYPE OF PROTECTION: The program is a 4K cartridge program that resides at \$8000. It uses the CBM80 to provide an auto start. There are two locations in memory where the program protection is resident. The first location is at \$85AB, this is a direct addressing routine. This code will directly try to change the program as it runs. If the program is resident in ROM (cartridge) the it can not be changed.

HOW TO COPY TO DISK: Use the CARTRIDGE CRACKER (TM). The CARTRIDGE CRACKER is a very unique package that consists of an expansion board and software on a disk. This unit will allow the user to down load a cartridge to disk. The CARTRIDGE CRACKER will also remove the protection from the cartridge before it saves the code to disk. Now, the best part: the CARTRIDGE CRACKER will give you a report of what code it changed to make the program work from disk!! The report can be printed out to the screen or to your printer. This whole process takes less than five minutes from start to finish. Finally, the CARTRIDGE CRACKER will write an auto-boot routine that will load and execute the program from your disk. Total time: 5 minutes start to finish.

HOW TO COPY TO EPROM: The method described here uses the PROMENADE EPROM programmer.

- 1) LOAD "HIMON",8,1 then SYS49152
- 2) Turn your cartridge switch off. Insert your HESMON (TM) cartridge. Turn your cartridge switch on. You may now examine the cartridge.
- 3) Save to code out to disk: S "HESMON-8000",08,8000,9000
- 4) Transfer the code at the cartridge location (\$8000-\$9000) to \$2000 in the computers memory. It is not absolutely necessary to transfer the T8000 9000 2000
- 5) Save the code at \$2000 out to disk. S "HESMON-2000",08,2000,3000

- 6) You now have the cartridge program saved to disk. One program will load into memory at \$8000 and the other will reside at \$2000. They both contain that same code, the only difference is where they reside in memory. Now that you have saved the program to disk we will proceed to burn the EPROM.
- 7) Turn the computer off so that we start fresh. Insert the PROMENADE into the modem port. Turn the computer on. Load and execute the PROMOS SOFTWARE (included with the PROMENADE). Then zero the PROMENADE (press 'Z').
- 8) LOAD "HESMON-2000",8,1
- 9) Insert the EPROM (2732) into the socket on the PROMENADE.
- 10) To burn the EPROM use the following command:
(pi sign) 8192,12287,0,224,7 RETURN
- 11) In less than a minute you will have 'burnt' a working copy of the program to the EPROM. Insert the EPROM into a cartridge board and you're done.

HOW TO MAKE A WORKING COPY OF THE PROGRAM ON DISK.

The techniques used here are described in the chapter on cartridge programs in the Program Protection Manual.

- 1) LOAD "HIMON",8,1 then SYS49152
- 2) Turn your cartridge switch off. Insert your HESMON (TM) cartridge. Turn your cartridge switch on. You may now examine the cartridge.
- 3) Save the code out to disk: S "HESMON-8000",08,8000,9000
- 4) Transfer the cartridge memory from ROM to RAM. T 8000 9000 8000
This will make a carbon copy of the program in RAM.
- 5) Turn off the cartridge switch and remove the cartridge.
- 6) Transfer the memory from \$8000 - \$9000 to \$2000. T 8000 9000 2000
- 7) Change the CEM80 to CEM00 at \$8000. This will prevent the program taking control after a reset.
- 8) Execute the program by using the warm start vectors. G 8E72
- 9) The program will mal-function. Reset the computer and re-enter your HIMON by SYS 49152.
- 10) Use the compare command to find the memory that changed in the RAM version. C 8000 9000 2000

- 11) You will find that \$824D has been changed to \$02. Use your ML monitor to hunt for the code that loads the accumulator with #\$02 (A9 09 LDA #\$02).
H 8000 9000 A9 02
- 12) By examining all four locations that load the accumulator with #\$02 you will find the code near \$85A9 and 8D71 very interesting. Both locations are followed immediately by a store the accumulator to a location and a compare with that same location. Keep in mind that you can not store a value in a memory location that contains ROM. When a cartridge program stores a value to a location and then proceeds to compare the same location, it is a check to see if the program is in RAM or ROM.
- 13) Transfer memory from \$2000 - \$3000 to \$8000. This will be the original program, back in the original location. T 2000 3000 8000
- 14) Change the code at \$85AB - \$85AD to EA (NOP) and change the code at \$8D73 - \$8D74 to EA (NOP).
- 15) Save this code out to disk. S "HESMON-FIXED",08,8000,9000
- 16) YOU'RE DONE. Now, which do you think is easier, the CARTRIDGE CRACKER or the way outlined in the PROGRAM PROTECTION MANUAL????

P.S. Press 'RESTORE' to execute the program.

SPECIAL FEATURE FOR OWNERS OF PROMENADE.

For the advanced programmer who would like to enhance the capabilities of Hesmon (TM), we will describe the process of eliminating some of the resets below \$0400. This will allow you more flexibility in the examination of programs that store to the vectors normally reset by Hesmon. Change the code to the following:

```
8E07 A9 E7 85 01 EA EA EA EA
8E25 EA EA EA A9 8E EA EA EA
8E2A EA EA EA A9 01 8D 20 D0
```

Hesmon (TM), Copyright 1982, Human Engineered Software.

The program is IMPERIUM GALACTUM, Copyright 1985 by Strategic Simulations Inc.

TYPE OF PROTECTION: There is an error 23 on sector 33/5. The disk is software write-protected by placing an "E" (\$45) on sector 18/0, byte \$02.

HOW TO COPY: Any copy program that copies error 23's can be used.

HOW TO MAKE A WORKING COPY WITH NO ERRORS:

1. Format a disk and copy the files from the original disk to the formatted disk with a file copy program. Some file copiers have difficulty copying the boot file since its name contains unprintable characters (DI-SECTOR turned the file into a SEQ file - but it loaded just fine with LOAD>"BOOT,SEQ",8!). Try another file copy program. If you can't find one that works, try to skip copying the first file. After copying the rest of the files, you can transfer the boot file through BASIC. Put your original disk in and type LOAD "BOOT*", 8. Put the copy disk in and type SAVE "BOOT",8.
2. If nothing you tried in step 1 works for you, make a 3-minute copy of the original with no errors. Run the UNWRITE PROTECT program from the JUNE NEWSLETTER on the copy (or use the UN W/P option from DI-SECTOR'S format editor). Validate the disk.
3. Using either step 1 or 2, you should have 347 blocks free. The boot file loads a file called RTL-64 at \$A000. This is the Run-Time Library for the INSTA-SPEED compiler. The boot program contains a small compiled program that loads in the file called HELLO.
4. HELLO is a simple BASIC program. Line 0 has been modified to be unLISTable from BASIC. An ML monitor can be used to examine the program after it has been loaded. Don't try to LIST the HELLO program. Delete line 0 by typing a 0 and pressing RETURN. Now you can list the HELLO file.
5. At the very end of the HELLO file is the error checking routine. Yep, it's in BASIC. Track 33, sector 5 is checked for an error 23. If the error is found, the program branches to line 907. Otherwise, the program goes into an endless loop at line 906: GOTO 906. We can defeat the protection check by changing line 906 to GOTO 907. Now the program will always end up at 907, regardless of an error 23 or not.
6. Save the modified program onto the copy disk:

SAVE "@0:HELLO", 8

(If SAVE@ makes you nervous, scratch the file first and then save it. See the article on page 7 about the SAVE @ bug)
7. That's it - you're done. The program will boot as usual.

This program is a curious mixture of new and old. There's an old-fashioned error 23, but the head knock is disabled first. Some programs are in BASIC but others are compiled. The program has an automatic fast load routine, for 1541 drives only.

The program is JUPITER MISSION 1999, Copyright 1985 by The Avalon Hill Game Co.

TYPE OF PROTECTION: A special DOS routine is used to read and decrypt a piece of program code from "under" a custom-formatted track.

HOW TO COPY: This disk could not be copied by most of the leading nibble copiers. Some of the new parameter copiers may be able to reproduce this disk.

HOW TO MAKE A WORKING COPY WITH NO ERRORS ON THE DISK:

1. Copy both sides of the original disk using a fast copy program which does not put any errors on the disk.
2. Run a track & sector editor. Insert side 1 of the COPY disk into the drive (all changes will be made to the same side of the copy disk).
3. Read in track 18, sector 9. Change byte \$7E (byte 126 in decimal) from a \$4C (76) to a \$00. Save the block back to the copy disk.
4. Read in track 18, sector 12. Change byte \$E4 (228) from an \$A2 (162) to a \$00. Save the block back to the disk.
5. Reset the computer or turn it off and on. With side 1 of the COPY disk still in the drive, type:

```
LOAD":*",8,1      and hit RETURN
```

6. The drive should come on for a bit, and then the computer will go back to "READY". Now, remove the copy disk and insert side 1 of the ORIGINAL disk in the drive. Type:

```
SYS 49152      (RETURN)
```

7. The drive will come on for a while again and then the computer will go back to "READY" again. Remove the original disk and insert side 1 of the COPY disk. Don't execute any commands such as loading the directory - there is a BASIC program in memory that you don't want to disturb. Now enter the following commands:

```
OPEN 15,8,15
PRINT#15, "V0"      (this will take a while)
PRINT#15, "S0:JMI999"
SAVE "JMI999",8
```

8. That's it - you're done. Just load and run the first file on the disk to start the game.

Okay, let's see how this program works. The boot program is called JMI999 and it loads in at \$0102-01FD, in the stack area of memory. It executes itself automatically at \$0102 after being loaded (see PPM Vol. II for an explanation of how stack autoboots work).

The boot uses an interesting way to load in the next piece of code. You're probably familiar with using an "*" as the file name when loading from disk (e.g. LOAD"*,8,1). Usually, this command will load in the FIRST file on the disk. Under some conditions, however, the "*" will cause the PREVIOUS file loaded (if any) to be reloaded instead. This happens if you haven't changed disks or reset the drive since the last time you loaded a program (By the way, you can prevent this by using ":"* instead of "**")

The boot uses the "previous file" feature of "*" to load in a piece of code that isn't contained in any file. The location of the first block of the most recently loaded file is stored in two locations in the drive's memory. The track is stored at \$7E (called PRGTRK) and the sector is stored at \$026F (PRGSEC). The boot uses memory-write (M-W) commands to change these locations to point to track 18, sector 12. This sector is linked to another block at 18/15 just like normal program file blocks are linked together. After setting the track and sector, the boot program performs the equivalent of a LOAD"*,8 from machine language. The two-block file is loaded into memory at \$C000-C186.

Normally, the code at \$C000 (49152) is executed after it is loaded in. In step 3 above, we change a JMP \$C000 statement in the boot into a BRK (\$00). When you run the modified boot in step 5, it loads in the code at \$C000 but the BRK sends the computer back to READY instead of executing the code. This gives us the chance to insert the original disk before running the \$C000 routine (step 6). The \$C000 routine begins by executing a special DOS routine in the drive, using a block-execute (B-E) command. The DOS routine is located in block 18/18 and is executed at \$0400 in the drive. This routine does more than just return a single protection value - it retrieves an essential piece of code from "under" the custom formatting on the disk.

The first block of this code is passed back to the computer and stored at \$C300. The code at \$C300 is then decrypted and transferred down to the normal BASIC area at \$0801. The process is repeated for another block of code, which is added onto the first part. After all this, the code at \$0801 turns out to be a short boot program in BASIC! A routine at \$A533 in the BASIC ROM is called next to link the BASIC program lines together. Then a RUN command is put into the keyboard buffer and the BASIC system is started up through its warm-start vector at \$0302. BASIC reads the characters in the keyboard buffer and performs a RUN command to start the BASIC boot. The BASIC boot sets the end of BASIC back to \$2100, which is necessary for the game to work correctly. Finally, it loads in the next program, INTRO, to begin the game itself.

In step 4 we put a BRK instruction (\$00) into the \$C000 routine at \$C0E0. This forces the computer back to READY after the BASIC boot is loaded, decrypted and linked together. At this point, the BASIC boot is sitting in memory like a normal BASIC program, ready to be saved to disk (control characters in line 2 prevent it from being LIST'ed). The validate command (V0) in step 7 frees up some space on the disk, and then the original JM1999 file is scratched and replaced with the BASIC boot. The boot file name must be named JM1999 because after a game is over, the program restarts itself by loading JM1999. The disk can be file copied now, and should run on non-1541 drives even if the original doesn't.

Each month we get a terrific number of requests for various programs. While we cannot honor each request for individual programs, we do try to find similar programs that may apply to many different requests. This month we will 'drop back' in time and review a number of the more 'basic' types of program protection schemes. These are the mostly the programs written in BASIC that contain bad blocks. Evidently there is still a large group of readers that are still having trouble with this type of program. Those of you that are more advanced may still want to refresh your memory with these types of schemes. If you are still having trouble with programs of this type refer to the PROGRAM PROTECTION MANUAL FOR THE C-64 (VOLUME I).

The program is **KIDS ON KEYS**, Copyright 1983, Spinnaker Corp., 1983.

TYPE OF PROTECTION: This program will check TRACK 2 SECTOR 15 for an error 23 and TRACK 3 SECTOR 16 for an error 27. Again we see your R/W head getting 'beat-to-death' more than once just to load the program.

HOW TO COPY: Copy the disk and place an error 23 on 2/15 and an error 27 on 3/16, or use a copy program that will place these errors for you.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

This program utilizes some of the early program protection techniques. The error checking is done through the boot program called "SPIN". "SPIN" is a basic program, but when we attempt to examine it with a normal list, we find one line (5) followed by a remark (REM). Cursor up to this line, press RETURN, and list the program. Everything appears to be fine until we reach line 110. From this point on we find that a portion of each line will disappear. This is a result of the programmer's use of multiple delete characters. This technique is covered in the Program Protection Manual Volume I. We can get a look at the program through LOMON. Using the I command at 0801 and following, will reveal the intent of the program. Notice the B-R (Block-Read) instructions that point to tracks 2 and 3. Another way to get a look at this program is to list it to your printer.

The programmer has gone to great lengths to protect his boot program, but left the main program virtually unprotected. Let's get to it. All we have to do is to lift a working version of the program from the computer's memory.

1. Make a copy of the program disk without errors. NEVER, NEVER ATTEMPT TO MODIFY AN ORIGINAL DISK!!!
2. Load and run the program in the normal fashion.
3. Once you reach the main screen, press the RUN/STOP and RESTORE keys.
4. List the program. Cursor up to LINE 1 and press the RETURN key. The entire program is now revealed.

5. You may now save the program to your copy disk with SAVE "KIDS ON KEYS",8.
6. To utilize the program, LOAD "KIDS ON KEYS",8 and RUN.
7. YOU'RE DONE!

The program is **KOALAPainter** (TM) by Koala Technologies Corp., Copyright 1983 by Audio Light, Inc.

TYPE OF PROTECTION: This program will check TRACK 33 for an ERROR 23.

HOW TO COPY: Use any copy program capable of placing an error 23.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

This is a program that will rattle your drive for absolutely no reason. The program is useless without the Koalapad hardware, and yet the programmer adds a needless protection scheme that will eventually throw your disk drive out of line.

If you examine the code from \$39AC through \$3A15, you will find that this program will read the error channel and store the values it finds at \$C7D7 and \$C7D8. It will check for the bad block first and return an error 23. Once the program passes this check, it will return to read a good block and store those values in the same location.

We will bypass the protection in the following manner:

1. Make a copy of the program without errors on the disk.
2. Load and run the program from the original disk. Once the program is running, RESET your computer.
3. Load and execute LOMON. Save out the code from \$C000 through \$C800 to your copy disk with S "@0:KPAINT1",08,C000,C800.
4. Now for the main body. Load the program from the original disk with L "KOALAPainter",08.
5. Using the M command at \$0A81, change the \$A0 to a \$60 (RTS).
6. Save out the altered code with S "@0:KOALAPainter",08,0900,4600.
7. To utilize this program follow the procedure below:

```
LOAD "KPAINT1",8,1
LOAD "KOALAPainter",8,1
SYS2304
```

Your swap screen will come up black with this program. Just use your X-COLOR option to change the screen to white. You may wish to write a boot program that will load the two programs and SYS to the location given.

8. YOU'RE DONE!

The program is **LEADER BOARD GOLF**, by Bruce and Roger Carver. Copyright 1986, Access Software Inc.

TYPE OF PROTECTION: This program depends on a **SECURITY KEY** or "DONGLE" for protection. The disk itself is file-copyable, but the program will not run unless the dongle is inserted in the cassette port. This is unusual since most dongles plug into one of the joystick ports (see "Millionaire" in the Feb. 1986 Newsletter). The original dongle is extremely simple compared to most dongles. Only two pins of the port are used, pins 5 & 6. These are the last two pins on the top right as you look at the port from the rear (plug's-eye view). On the original dongle, these two pins are connected by a 1 OHM resistor. That's all there is to it!! In fact, the resistor isn't really necessary - a piece of wire will do. Just don't plug the dongle in the wrong way or you'll probably blow the power supply (at least)!

HOW TO MAKE THE PROGRAM WORK WITHOUT A DONGLE:

1. Copy the files from the original disk or make a copy of the whole disk without errors. Scratch files "L" and "H" from the COPY disk.
2. Load and execute an ML monitor at \$C000 (49152), such as HIMON. The monitor must be able to work with the BASIC ROM switched out.
3. Insert the ORIGINAL disk and load file "L" using L"L",08 . Transfer a copy of the file up in memory with T 081D 3E32 481D . This is for a double-check later.
4. Change the following bytes as indicated (all values are in hex). Be careful!! A partner will make it much easier.

<u>LOCATION</u>	<u>ORIGINAL</u>	<u>CHANGE TO</u>	<u>LOCATION</u>	<u>ORIGINAL</u>	<u>CHANGE TO</u>
OCDF	36	26	2611	2F	3F
OCEA	35	25	26BC	2F	3F
OFB5	2F	3F	2719-1B	99 D1 FF	EA EA EA
OFB9	37	27	2857	2F	3F
OFC2	36	26	2947	2F	3F
1015	35	25	2BF8-F9	91 9C	EA EA
1126	2F	3F	2DF9	2F	3F
112C	F7	E7	2FCB	F7	E7
1436	2F	3F	3235	37	27
14FC	20	30	3238	0A	1A

158A	2F	3F	323B	0A	1A
192C-2D	85 00	EA EA	3327	99	B9
1B81-82	85 00	EA EA	3332	9D	BD
1F7A	2F	3F	3C42	2F	3F
236F-70	85 00	EA EA			

- When you're done, compare your new version with the old copy in memory using C 081D 3E32 481D . Make sure only the correct bytes were changed. Insert the COPY disk and save the new version using S"L",08,081D,3E33.
- Now you have to switch out the BASIC ROM. Display location \$0001 with M 0001. Change the \$37 to a \$36 and hit RETURN.
- Insert the ORIGINAL disk and load file "H" using L"H",08. Transfer a copy down in memory with T 9280 AB9A 7280 for double-checking.
- Change the following bytes.

LOCATION	ORIGINAL	CHANGE TO	LOCATION	ORIGINAL	CHANGE TO
93F0	10	00	A230-31	85 00	EA EA
98BC	2F	3F	A7CF	2F	3F
9BD4	2F	3F	A7D7	F7	E7
9F5A	2F	3F	A873	2F	3F
A1C5	2F	3F			

- Double-check the new version against the old one: C 9280 AB9A 7280 . Insert the COPY disk and save the file using S"H",08,9280,AB9B . That's "all" there is to it - you're done! (finally)

Well, what can I say? Do you really want to know what all that does? OK, OK. I won't have room to cover the changes individually this month, so we'll take another look at them next month. In the meantime, once you understand how the dongle works you'll understand most of the changes immediately. The dongle connects together the TAPE WRITE and SENSE lines on the cassette port. These lines appear in memory as bits 3 and 4 of location \$0001 . Bit 3 is WRITE, for writing data to tape. Bit 4 is SENSE, for sensing the tape PLAY button. Note that WRITE is an OUTPUT line (naturally) and SENSE is an INPUT line (like your senses). Now, as long as the dongle connects the WRITE output to the SENSE input, whatever value is written to the WRITE bit (0 or 1) will magically appear in the SENSE bit. For instance, if you write a \$37 value to \$0001, you'll be setting bit 3 (WRITE) to a "0" (\$37 = 0011 0111 in binary). You'll also be sending a "1" to bit 4 (SENSE) but since SENSE is an input this value is ignored. Instead, a "0" will appear in the SENSE bit, from WRITE. So even though you wrote a \$37, when you read it the value will be \$27 (0010 0111 in binary). Try it - plug the dongle in and try to set \$0001 to a \$37. You can't do it.

This is the basis of the dongle protection scheme. The program keeps setting WRITE to 0. As long as the dongle is in, SENSE will be 0 too. With no dongle, SENSE is always a 1 regardless of WRITE. If the program detects a 1 in SENSE, it

will crash. Simple enough scheme. When I realized how the dongle worked, I thought this program would be a snap. The problem is, the program has dozens of checks for the SENSE bit! I found most of them (I think) by reading through the files, but it only takes one to crash the program. Some were pretty sneaky (see \$14C8-E1, 1245-4E and \$3A11-25), and I began to despair after a couple days. Then I thought of another way to go. There IS a way to set SENSE to a 0 WITHOUT the dongle - IF you make SENSE an OUTPUT rather than an INPUT. OUTPUT lines always hold the last value written to them. The lines in \$0001 can be selected for INPUT or OUTPUT through location \$0000. If a bit in \$0000 is a "1", the corresponding bit in \$0001 will be an OUTPUT. A "0" will give an INPUT. The normal value in \$0000 is \$2F (see pp. 260-1 of the Prog. Ref. Guide). Changing the \$2F to \$3F will change SENSE to an OUTPUT. The program sets \$0000 to \$2F many times, so these all had to be changed to \$3F's. Also, any time \$0001 is written to, we must make sure the SENSE bit gets the same value as WRITE (0 in all cases). That accounts for most of the other changes. Finally, there's also a couple checks to make sure SENSE is selected for INPUT. Whew!

LEADER BOARD GOLF, PART 2

Last month we presented the backup procedure for LEADER BOARD GOLF. However, there were so many changes to make that we didn't have room to explain them individually. This month we'll explain the changes, but first let's review the way the protection scheme works. The protection scheme uses a security key or "dongle", which plugs into the cassette port. The dongle consists of a small resistor connected between the TAPE WRITE and TAPE SENSE lines. It is essential to remember that WRITE is normally an OUTPUT line and SENSE is normally an INPUT line. With the dongle plugged in, a bit sent out via the WRITE line is automatically transferred to the SENSE input, where it can be read by the computer.

As far as protection is concerned, however, all bits are not created equal. The reason is that WITHOUT the dongle, the SENSE input always has the value 1. Therefore, sending a 1 bit to WRITE and checking for it to appear in SENSE would not really check for the presence of the dongle. Only sending a 0 bit to WRITE would be a true test; this is the only way to make the SENSE input yield a 0 value. This is precisely how the program checks for the dongle. It sends a 0 bit to WRITE and checks for it to appear in SENSE. The strength of the protection scheme lies in the fact that it does this not just once but many times. My first attempt to disable the protection was based on finding all these checks. While I found quite a few, there always seemed to be another one left somewhere (you might want to try your luck at finding them yourself).

Then I tried another approach. Rather than disable the checks, I realized there was another way to make a 0 appear in SENSE. The situation described above only applies if SENSE is an input. However, like the other I/O lines in the microprocessor (and CIA chips), SENSE can be set for input or output. When an I/O line like SENSE is set for output, it retains the last value written to it. If we make SENSE an output and set it to 0, it will keep that value and thus pass all the protection checks. Therefore our job is two-fold - keep SENSE set as an output and keep it set to 0. Our task is again complicated by the fact that the

program tries to make SENSE an input many times; it also checks to make sure SENSE is an input and tries to set it to 1 several times. Fortunately, I was able to find all of these using a "brute force" approach - reading through nearly 20K of ML code and making notes as I went. Let me tell you, if it was just for me, I would have given up the first day. Only the thought of all you poor Newsletter subscribers out there kept me going (how about returning the favor by re-subscribing?).

To get down to specifics, we need to see how to make SENSE an output and how to set it to 0. The SENSE (and WRITE) lines appear in location \$0001 in the computer, the same location used to switch the BASIC and KERNAL ROMs in or out. The SENSE line is bit number 4 of this location (bits are always numbered 76543210 in the byte). This results in an easy way to tell what value corresponds to the SENSE bit: if the first hex digit (high nibble) of location \$0001 is even, SENSE is a 0. If the value is odd, SENSE is a 1. Since we want SENSE to be a 0, we want to make sure the high nibble is even. Now pay attention, because this can be confusing. To make SENSE an output, we have to manipulate a different location, \$0000. Specifically, we have to set bit 4 of this location to a 1. Therefore, we want the high nibble of location \$0000 to be odd, just the opposite of what we want in location \$0001. Got that straight? Good.

OK, now we're ready to look at the code and the changes we made. Fortunately for all of us, many of the changes were the same (please dig out last month's issue for reference). Let's start with the most common one. Surely you noticed how many times we changed a \$2F to a \$3F. The program stores a \$2F in location \$0000 before checking the SENSE bit in \$0001. The \$2F value makes SENSE an input; by changing it to \$3F we make SENSE an output (note that we are making the high nibble of location \$0000, as we said we wanted to). There are some other places where a \$2F is stored into \$0000 (via STA>\$00 statement) that we made a different change. When this type of code appears at the end of a subroutine, the value \$2F is normally left in the accumulator (A). After calling the subroutine, the program could potentially check for the value \$2F in A. Therefore, to be ultra-careful, we leave the \$2F value alone. Instead, we replace the STA \$00 statement with two NOPs. This accounts for the changes from \$85 00 to \$EA EA (the reason we didn't make this change every time was to reduce the number of bytes changed, making the backup procedure easier to perform and check).

The program still has several tricks up its sleeve for putting a \$2F into location \$0000 and checking it later. Rather than do it directly, it can be hidden using math functions and indexed addressing:

```
14F9 LDA #$0F
14FB ORA #$20      A becomes $2F - change $20 to $30 for $3F.
14FD LDY #$01
14FF STA $FFFF,Y  Stores A into $FFFF+01 = location $0000!

2715 LDA #$5E
2717 LSR           Same as divide A by 2 - equals $2F.
2718 TAY           Y = $2F too
2719 STA $FFD1,Y   $FFD1+2F = location $0000; replace with NOPs ($EA).
```

This same code appears at \$3323-29. In that case, it's not the end of a subroutine and the value in A is wiped out immediately. Therefore, we could replace the STA \$FFD1,Y with three NOPs. However, it only takes one byte to change the STA to LDA (change loc. \$3327 from \$99 to \$B9).

```

2BF4 LDA #BC
2BF6 LSR          Divide A by 2 = $94.
2BF7 LSR          Divide again = $2F.
2BF8 STA ($9C),Y  $9C and Y are $00's = loc. $0000; change to NOPs.

93EF LDA #10      Change $10 to $00 to give correct result.
93F1 BIT $00       AND's loc. $0000 with A - tests if bit 4 is a 1.
93F3 BNE $93F6     Branch if SENSE is an output (bit 4 = 1) - crash!

A22B CMP #2F      Check key pressed - $2F = "/" (restarts game)
A22D BEQ A230
A22F RTS
A230 STA $00       Use $2F value to restore loc. $0000 - change to NOPs

```

Note how tricky this last change is - if we had simply changed \$2F to \$3F, the restart option would be affected.

That's it for the original code that sets SENSE to an input and checks it. The changes we made will make SENSE an output instead, and disable the two checks (at \$93EF and \$A228).

Now for the changes to set SENSE to a 0. Remember, to do this we want the high nibble of location \$0001 to be even. As we said, this same location is used to switch out BASIC, the KERNAL or the I/O devices (VIC, SID, CIA's). Probably the most familiar case is switching out BASIC by changing this location from \$37 to \$36 (we did this in the procedure last month). Note that this value has an odd high nibble (3). This would set SENSE to a 1, so to set it to a 0 we have to use the value \$26 (C128 owners take note: the high nibble is normally a 7 rather than a 3, but setting it a 2 will still be OK). This accounts for all the cases (except one) where we changed a \$37, \$36 or \$35 to a \$27, \$26 or \$25. The same logic accounts for the places where we changed a \$F7 to an \$E7. In those cases, the program is AND'ing location \$0001 with \$F7, which preserves the value in SENSE (bit 4). We want to make sure SENSE is actually set to 0, so we AND with \$E7 instead. The result is stored back to \$0001. This is also what is happening in our last piece of code:

```

3232 LDA #C0
3234 AND #37       A = $F7; change $37 to $27 to get $E7 instead.
3236 TAX          X = $F7 too (or $E7 after change).
3237 AND $FF0A,X   $FF0A+F7 = loc. $0001; change $0A to $1A (both
323A STA $FF0A,X   places) to get correct result with $E7.

```

Similar code occurs at \$332A-34, except \$07 and \$F0 are AND'ed to get \$37, rather than \$C0 and \$37. There we simply disabled the STA>\$FF0A,X by changing it to LDA.

Well, those of you that are still awake have seen a lot of clever things. I've got one more fascinating tidbit to pass along. The protection scheme depends on the dongle being (almost) the only way to make SENSE be a 0. Consider the SX64 portable, however. This machine does not HAVE a cassette port, so what do you do if all you have is the original version? Well, it turns out that SENSE is a 0 on this machine normally, so the original program runs WITHOUT a dongle! That's right - you don't have to change the program at all! The modified version we've given you will also work on the SX. By the way, there's reportedly another version out there, so be prepared. You may be able to find the same code in different areas and modify it the same way. Good luck!

The program is **LODIE RUNNER** (C) 1983 BY BRODERBUND

TYPE OF PROTECTION: The three main programs are stored on the disk in program files. The boot program (LR) is written in BASIC. This program will poke a ML program into the screen memory and execute the ML program. The reason that you do not see any ML program on the screen is that the screen color (line 100) has been set to black and the character color ram also has been set to black in line 1070. This ML routine contained in screen memory will load the other two programs from the disk then jump to the proper entry point of the main program. The main program (IT) contains the program protection scheme. If you use you ML monitor to examine the main program you will find that the program has two areas in memory where the KERNAL calls and 'U1' reside (\$723E and \$8EB6). Keep in mind that the program contains a game generator that allows you to make your own screens and that the program also contains many different screens stored on the disk. The different screens that are stored on the disk do not appear in the directory. This means that they are loaded in from the disk from user files. Hence, it would appear that one area of the program will be used to save the various screens that the program uses.

HOW TO COPY THE PROGRAM

Due to the combination of errors on the disk, the program may be difficult to copy with the normal means. The error combination may be duplicated with some difficulty by simply copying the disk and placing the correct errors on the copy disk. The program may also be copied by many of the better nibble copy programs.

A word on nibble copy programs is in order. The term nibble copy comes from the copy programs that are used on the APPLE (R) computer. A nibble is 1/2 of a byte or four bits. These copy programs will copy the disk four bits at a time, making an exact copy of the source disk. While the copy programs for the 1541 do not actually copy four bits at a time, the name for some of the better copy programs has carried over.

HOW TO MAKE A WORKING COPY WITHOUT ANY ERRORS ON THE DISK

The techniques used on this program are described on pages 50-60 and on pages 71-79. Plus, a new twist will be introduced in this protection scheme. You will be able to apply the information contained here on many different programs.

- 1) Copy the original disk with any good copy program that will not place any errors on the destination disk. Load the directory and make a note of how long each program is.
- 2) Load and execute the HIMON. Fill in memory from \$0800 to \$BFFF with \$00. This will erase any random memory from the computer. Load the program called IT from the copy disk. This is the program that contains the protection scheme. Use the H command to hunt for the 'U1' and the KERNAL calls. Write down the locations of the U1 for future reference (\$723C & \$8EB6).
- 3) Keep in mind that program called IT was 89 blocks long. Use your monitor to find the beginning of the program (\$6000). Roughly calculate the ending address of the program (4 blocks = 1K) and you will find that the program should end between \$B000 and \$C000. In order to find the exact ending address of the program it will be necessary to turn off the BASIC interpreter and examine the RAM that underlies BASIC. To do this use the M command of your monitor.

.M 0000 (RETURN)

.:0000 2F 37 C0 3E 32 00 C3 00

Change the \$37 contained at location \$0001 to \$36. This will turn off the BASIC interpreter and allow you to examine the RAM that underlies that area of memory.

- 4) Again use the H command to hunt through memory for the 'U1' and the KERNAL calls. Verify that there is not any other area of memory that contains KERNAL calls. Now it will be necessary to examine each part of memory where the kernal calls reside. Use the I command to examine memory in the general location of where the U1 resides (\$7000-\$7400 and \$8D00-\$8FFF). It will be necessary to rule out one area of memory as the screen loader and concentrate on the other area of memory as the protection scheme. The USER (U1) at \$8EB6 points to the areas of this disk where the errors reside (Track 1) so lets concentrate on this area as the protection scheme.
- 5) Disassemble the code in this area and comment it fully in order to better help you understand the logic of the program.
- 6) The code starting at \$8E44 appears to a program protection scheme (see page 60 P.P.M.) except for the comparison and branch instructions. This code only checks the error channel and stores the error code returned from the disk

drive in memory (from \$8E8D to \$8E9A). Once the error codes are stored in memory (from \$8E3C to \$8E43) the program will execute. The main program will check memory to see if the proper errors were found and stored in memory. If the proper errors were not found, the program will not operate properly. Now that you have a better idea of the protection scheme let's get to work!

- 7) This program uses both PROGRAM and USER files on the same disk. It will be necessary to take extra precautions when saving the code to the copy disk. It will be necessary to allocate the entire BAM and then scratch the file IT from the copy disk. This way when you save the modified code out to the disk there will not be any chance of over-writing any USER file. To allocate the Bam it will be necessary to use a TRACK and SECTOR editor (DISK DR, etc.). Fill the BAM with 00's (bytes 4 thru 143, tr 18, se 0). Then exit the TRACK and SECTOR editor. Scratch the file IT from the disk (OPEN15,8,15,"SO:IT"). Due to the many user files on the disk do not save any other programs on this disk.
- 8) Load and execute the original program normally. After the program is in memory and running, RESET the computer. Load and execute the HIMON. Turn off BASIC (as described above) and examine the code from \$8E3B to \$8EA0. Notice that the program stored the specific error code in memory that the program wants to see (locations 8E3C to 8E43). Change the code at \$8E92 to \$8E9A as follows.

8E92	EA	NOP
8E93	EA	NOP
8E94	EA	NOP
8E95	20 CF FF	JSR
8E98	EA	NOP
8E99	EA	NOP
8E9A	EA	NOP

- 9) Save the code back to the copy disk. S "@0:IT",08,6000,B8000 (RETURN)
- 10) YOU'RE DONE— Remember what you did here, you may run into similar schemes!!!

Amendment to LODER RUNNER (C) 1983 BY BRODERBUND.

Some people have had trouble with this program. I will give you an alternative path to follow.

- 1) Make a copy of the original disk without any errors.
- 2) Load and execute the HIMON. Load the program called IT from the copy disk. Modify the program as follows: Use the M command to change the following 8 bytes. M 8E3C : 8E3C 32 30 32 30 33 30 37 30
- 3) Change the code from \$8E92 - \$8E9A as before.

- 4) Turn off BASIC and save the program out to the copy disk.
S "@0:IT",08,6000,B800
- 5) YOU'RE DONE

If this does not do it for your program they may have changed the protection scheme, again.

The program is **MAGIC DESK 1 (TM)**, Copyright 1983, Commodore (R).

This program comes only on cartridge. It is not any ordinary cartridge, but a multiple ROM using bank switching as discussed on P.70 of the P.P.M.. There are four 8k ROM chips and two small switching chips inside. If you own this cartridge, you will notice it gets rather warm after running a while. Having a disk copy to back-up the files you create with it, could be a real asset should the complicated cartridge break down. A disk copy will not load as fast as the ROM's do, but will do everything else equally well.

When power is first applied with the cartridge in place, it starts out just like the 8k cartridges with the computer seeing only one 8k ROM located at \$8000 to \$9FFF, or 32768-40959 decimal. The cartridge places a ground on the EXROM load, and the start of the program has the familiar CBM80 auto-start, which is discussed on P.63 of the P.P.M.. The program begins executing at 32777 and after clearing the screen and initializing, it sets the top and bottom of memory for 2560 and 28592. This turns out to be the storage area for a very long program written in BASIC. Next, a routine at 32965-32989 is written to 2048 and executed. This is the key to reading the programs out of the other ROM's and storing them in RAM. An extra lead to the cartridge at pin 7 is used to detect pokes to memory location 56832. By poking different numbers to 56832 and using the subroutine at 2048 four times, the contents of all ROM's are read into RAM. The last half of the original 8k ROM that was present at power-up is also written up to occupy the 49152 area and this becomes the area of many short ML routines called by the big BASIC program below. The other ROM's all contain the long BASIC program. One other thing contained in the first ROM is the sprite table. The table begins at 32990 and each sprite is 64 bytes long. The first sprite is the telephone that you see on the desk. There are lots more. We counted 26 of them, more or less. We say that because we found that some of them are kept a secret by Commodore (R). They are the best sprites you'll never see. You won't see them because they are never called in the program. There's a house, a printing calculator, some strange cane-like objects, and a set of test tubes. Perhaps they are for some project Commodore (R) never finished or is still working on. Maybe in the future we can make test tube babies with our computers. The other sprites are for the icons you move and each item in the 4 HELP PAGE DISPLAYS is a sprite. The desk itself is drawn at the start of the BASIC program with ordinary PRINT statements.

Since only 8 sprites at a time may be used, the program picks the 8 it wishes to use at any one time and places them down at 2048-2559. This replaces the ROM loading routine, which is no longer needed. As different screens are

drawn, different sprites are fetched from the master table. The routine that does this is located at 49218-49333 in the running program, which was in the cartridge 12288 bytes below that. That is important to remember, since any changes to the ML must be made down there, so they will be written up to this location.

Once the Magic Desk (TM) is running, the only function of the cartridge is to supply the master sprite table as needed. The other ROM's are never accessed again. The rest of the original ROM has been written up to 49152, except for the sprites and the opening ML initialization. You could almost get away with turning off the power to the cartridge and saving the computer's whole memory, except for one thing. If you try turning off the cartridge power, you will at first see no change, but change screens and the sprites are gone, replaced by 3 vertical lines, once the cartridge is removed. If you were to examine the sprite master table area in memory, you would find endless 160's have been written to the entire area, covering everything. When we first saw this, we thought it was a copy protection scheme. When you type a page using the program, where do you suppose it is stored? It starts storing at 28672 and goes all the way to 33951. A typed page has 80 spaces across and 66 lines up and down. The program must be able to store those 5,280 possible characters. The 160s are their code for clear space. The 160s will be replaced by your typing as you fill the page. The only answer is to relocate that master sprite table and modify the ML to look for it in its new location. Now, where do we have 1600 or so free bytes to do that? After a thorough search, we find there is no place that long, so even more changes will be needed.

If we take a careful look at the big BASIC program, we find it begins at 2560 and ends at 21833, with the area from 21833 up to 28592 reserved for the variable and string storage any BASIC program needs. But that is an awful lot of storage space, maybe it could do well on less. In the ML initialization, the pointers for memory top and bottom are set, so lowering the top could provide the space needed to relocate the table. If we do so, then all the pieces fit into place. A good running copy is produced, which may be loaded and saved as any ordinary program on any disk. It does use 153 blocks and has a 2 minute load time, but once running, it is just like the cartridge in speed and features. The method to do the save is much simpler than the explanation.

MAGIC DESK I (TM) - CARTRIDGE TO DISK PROCEDURE

- 1). You will need a blank formatted disk, the cartridge, your cartridge switch (P.P.M. Page 62), or your Cardco 5 expansion board.
- 2). Type in and save the following BASIC program to the disk. Call this program "MODIFYDESK".

```
1 POKE 36969,106:POKE 36963,0
2 POKE 32804,0:POKE 32806,106
3 FOR A=32990 TO 34461:POKE A-5854,PEEK(A):NEXT A
4 FOR A=32864 TO 32908:POKE A,234:NEXT A
5 PRINT "MOD DONE"
```

- 3). Insert the cartridge. Turn on the computer, with your cartridge switch on (Cardco - right switch). Wait until the Magic Desk 1 (TM) main option screen is displayed. This is the screen after the copyright screen. We are assuming that you have verified that the cartridge works properly.
- 4). Turn off the cartridge switch (This is the right switch on the Cardco 5). The screen should not change.
- 5). Push the reset button. The normal blue screen should come up.
- 6). Turn on the cartridge switch (Cardco-right switch). The screen should not change.
- 7). Verify that the program is really there by entering PRINT PEEK(32772). You should get 195. That is the C in the CBM80 auto-start routine.
- 8). Enter these pokes in direct mode; POKE45,0:POKE46,159
- 9). SAVE "ORIGINAL",8,1. This will take about 2 minutes. If you should get graphics characters during the save process, you have not followed the procedure as described. Power-down and begin again. When the save is completed, turn everything off and remove the cartridge. Turn everything back on.
- 10). LOAD "ORIGINAL",8,1
- 11). Verify that the program is there with PRINT PEEK (32772). This should be done in immediate mode. As with STEP 7, you should get 195. Now POKE 32772,0. This will alter the CBM80, preventing the auto-start. We will leave it at 0 permanently.
- 12). Push the reset button. The computer should not change much.
- 13). Type in the following in direct mode: POKE 51,0:POKE 52,10:POKE 55,0:POKE 56,10 - These pokes will keep the desk program protected, as we load another program under it in memory.
- 14). LOAD "MODIFYDESK",8. You may list it to verify it is correct.
- 15). Enter RUN. The program will take 15 seconds to run and the screen will not change.
- 16). Push the reset button again.
- 17). Enter the following one line program, but DO NOT RUN IT. :
1 SYS 32786
- 18). POKE 45,0:POKE 46,159 (IN DIRECT MODE)

- 19). SAVE "MAGIC DESK",8,1 to the disk. This is the working version. After the save is completed, you may save it again to as many disks as you like with only the SAVE command. When you are through saving, enter RUN and it should run. At any future time, you may load it and immediately save it to another disk as well. Once loaded, type list and you should see the following: 1 SYS 32786. Once it is run, you may exit the program with the reset button, but it may not be started again. This is because the initialization is over-written by the 160's. This occurs after the need for that part of the program is finished. Once you have a working version, all the other programs saved to the disk may be erased.

FUN WITH THE DESK

Would you like to read the long BASIC program that makes the desk work? After loading it, enter POKE 44,10 and then LIST. Don't try to change lines or run it, because there are not enough changes to make that work, but you may read the 494 lines of BASIC as many times as you like.

One of the most interesting things you may do is to use the desk as a showcase for your own custom sprites. Many of the icons on the desk have no function, like the telephone, calculator and ledger, so they could be replaced with your own. We also could move some of those secret sprites up where they may be seen. Since we had to learn all about the sprite table to modify the program, we have made a list of the new sprite locations, after the modification. After loading Magic Desk I (TM), you may PEEK and POKE all the changes you like. You may also remove any of the icons to use in some other program or game. The area from 2048 to 2559 is free to load programs to POKE in sprites, as long as you remember to protect the area above with POKES to 52 and 56, as in the modifying procedure. That protects the variables from any short program and will prevent damage to the program above 2560.

Let's use an example of how to see these secret sprites. First, load the program and then do a reset or a NEW. Now enter the POKES from instruction 13 (POKE 51,0:POKE 52,10:POKE 55,0:POKE 56,10). Lastly, type in the following program:

```
1 FOR A =34590 TO 34653:POKE A-6366,PEEK(A):NEXT A
2 FOR A=34526 TO 34589:POKE A-7326,PEEK(A):NEXT A
3 FOR A=34462 TO 34525:POKE A-7134,PEEK(A):NEXT A
4 SYS 32786
```

Now when you type RUN, there will be a delay as the sprites are copied to their new locations and then the desk will come up, with a house in place of the ledger (line 1), a set of test tubes in place of the calculator (line 2), and a printing calculator in place of the card file (line 3). These 3 sprites were moved from their original location and were not relocated in the table below.

In a similar way you may change around any sprites in the program. Be aware that some sprites are expanded horizontally (like the hand) and others are double both ways (the ledger), while others are not expanded at all. As a result, the

sprites will look different in different places. The hand will look very small in place of the telephone. Here are the sprite data locations:

- 27136 - telephone on desk
- 27200 - calculator on desk
- 27264 - printer in typewriter mode
- 27328 - card file on desk
- 27392 - the pointing hand
- 27456 - the sleeve of the hand
- 27520 - wastebasket
- 27584 - desk symbol seen in typewriter mode
- 27648 - carriage point in typewriter
- 27712 - file folders symbol
- 27776 - a cane-like object
- 27840 - another cane-like object
- 27904 - page symbol that hovers over desk
- 27968 - another page symbol used in drawer
- 28032 - 3 page symbol
- 28096 - floppy disk symbol (a good one too)
- 28160 - error symbol
- 28224 - ledger on desk
- 28288 - file drawer seen in help screens
- 28352 - typewriter seen in help screens
- 28416 - box for the clock seen in help screen
- 28480 - exit sign seen in help screen
- 28544 - margin set symbol seen in help screen

Each sprite begins at the memory location and is 64 bytes long. There are 3 more sprites in the original table as discussed above, at 34590, 34526, and 34462.

A FEW NOTES

The modification routine was fairly well explained but a few extra points are in order. You will note that while the original began executing at 32777, the new version begins at 32786. This is to avoid a lock-up, which occurs when an initialization kernal routine is called. Anytime you find JSR 64848 or JSR 65415 at the start of a cartridge program, get rid of it in your RAM copy or skip it. There is no need for it anyway in a disk copy, because this has already been performed. This routine tests RAM and it appears that when it reaches itself in memory, a lockup occurs unless SEI has occurred. In fact, most kernal calls at the start may be skipped, since their purpose is moot.

Line 4 of the mod program removes the part that loaded the other chips. If this were not done, garbage would be written over the program already in memory. The part that moved the original section up to 49152 remains. The reason they move it is because the area from 34816 to 40096 is used to store a page loaded from the file cabinet, more of those 160's again. We found that out the hard way, when we first tried relocating the sprites up there and got clobbered again with 160's.

Line 1 sets the program to look for its sprites at page 106, which is 27136 and line 2 makes sure the variable storage ends at page 106. Line 3 moves the sprite table down from 32990 to 27136, and copies it there, 5854 bytes lower. Before you run the program, both tables will still exist.

Here is a partial memory map of what is going on when Magic Desk 1 (TM) is running.

2048 -2559 - Storage area for 8 sprites in current use
2560 -21833 - a 494 line BASIC program
21833-27136 - variable and string storage for the BASIC program
27136-28607 - master sprite table
28672-33951 - the page you type in the typewriter
33951-34816 - appears not to be used, but don't bet on it
34816-40096 - a page loaded from the file cabinet
49152-53247 - various ML routines called in the BASIC program

The program is **MASTER OF THE LAMPS** (TM) by Russell Lieblich & Peter Kaminski. Copyright 1985 Activision Inc.

TYPE OF PROTECTION: This program will check track 36 for an error.

HOW TO COPY: Some of the new nibble copy programs such as Mr. Nibble (TM) or Fast Hack'Em's SD2 V3.0 (MSD only) will copy this disk.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

Once this program has loaded the main file, it will check track 36 for the "KEY" value. If found, the program will execute properly. All we have to do is allow the program to execute, perform a reset, save out the code, and find an entry point.

1. Load the program from the original disk. Once the main title screen is up, RESET your computer.
2. Load and execute HIMON. Flip out BASIC by changing the byte at \$01 from a \$37 to a \$36.
3. Save out the main section of code to a newly formatted disk with S "MASTERMAIN",08,0800,BFFF.
4. We're not done yet. There is a section of code stored in the RAM under the KERNAL (\$E000-FFFF) that we must capture for proper execution of the program. We will use the procedure described in the November 1984 Newsletter to capture this code. Those who saved this routine need only load and execute it with G 1000. Those who have the PROGRAM PROTECTION MANUAL II should load "MOVE

KERNAL", execute it with G 1000 and then proceed to STEP 5. Those who do not have these programs should follow the procedure below:

You should still be in HIMON. Using the F command, fill the memory from 1000 to 9FFF with 99. This will clean things up for our next operation. We will construct the MOVE KERNAL routine at 1000. Using the M command, type in the following:

```
..:1000 78 A9 35 85 01 A9 00 85
..:1008 FC 85 FE A9 E0 85 FD A9
..:1010 20 85 FF A0 00 B1 FC 91
..:1018 FE C8 C0 00 D0 F7 E6 FF
..:1020 E6 FD A5 FD D0 ED A9 37
..:1028 85 01 58 00 00 00 00 00
```

Make sure you have typed in the code correctly. Don't forget to type return after each line. When we disassemble the program at \$1000, we find the following:

1000 SEI	100F LDA #\$20	101E INC \$FF
1001 LDA #\$35	1011 STA \$FF	1020 IND \$FD
1003 STA \$01	1013 LDY #\$00	1022 LDA \$FD
1005 LDA #\$00	1015 LDA (\$FC),Y	1024 BNE \$1013
1007 STA \$FC	1017 STA (\$FE),Y	1026 LDA #\$37
1009 STA \$FE	1019 INY	1028 STA \$01
100B LDA \$E0	101A CPY #\$00	102A CLI
100D STA \$FD	101C BNE \$1015	102B BRK

To get to the RAM under the KERNAL, we must switch off the KERNAL ROM. But first we must set the interrupt, which will keep the IRQ interrupt from trying to access the KERNAL. Then we simply transfer the code stored under the KERNAL to \$2000, turn the KERNAL ROM back on and clear the interrupt. Save this program to a utilities disk using S "MOVE KERNAL",08,1000,102C and keep it for future use. Now execute the program with G 1000. Using the I command, scroll through the code at \$2000 now to verify that it was transferred.

7. Save the code from under the KERNAL, now at \$2000, to your formatted disk with S "MASTERKERNAL",08,2000,4000. We may now exit the monitor.
6. We have a slight problem with our MASTERKERNAL code. When loaded, the file will locate itself at \$2000. The program is expecting to find it at \$E000. We could write a loader program to relocate this code, or we could do it the easy way. With a track and sector editor, we can locate where the first block of the MASTERKERNAL program is stored on the disk. Go to TRACK 18 SECTOR 01 and start looking through the directory for MASTERKERNAL. If you used a newly formatted disk, it will be the second program in the directory. Bytes 03-04 of the directory entry (counting the first byte as 00) will tell you where the first block of this program is located. On a newly formatted disk bytes 03-04 should have \$13 00. Once you have the location, go to that block (19 in this

case, since \$13=19). Byte 03 of the block will contain the high byte of the load address, which is \$20 for this program. If we change this to \$E0, the program will load at \$E000. The block should read as follows:

13 0A 00 20

13 0A - Location of the next block of the program (may vary).
00 - Low byte of the load address.
20 - High byte of the load address. Change the \$20 to an \$E0.

7. To execute the program use the following procedure:

LOAD "MASTERKERNAL",8,1
LOAD "MASTERMAIN",8,1
SYS 34305

8. YOU'RE DONE!

As you can see, the entry point we chose brings up the main title screen. We found this entry point by hunting for a store to border color (H 0800 BFFF 8D 20 D0). This hunt returned the address \$8611. Through the disassembly of this code, we determined \$8601 (=34305) as the point to try. Check the PPM II for a complete discussion of entry points.

The program is MBA-TOR(tm) copyright 1984 by AEA (version 27.DEC.84)

TYPE OF PROTECTION: This is a 16K cartridge program that uses indirect, indexed addressing - STA (88),Y. This form of addressing uses a pointer on zero page, and the value contained in the Y-register, to calculate the memory location to be acted upon. This cartridge is an AUTO-START cartridge, but it does not use the common CBM80 as found in most other cartridges that reside at \$8000-\$BFFF. This auto-start feature will be looked at in detail in just a few minutes.

HOW TO BACK UP: Although other versions of this program may be directly backed up to disk with the CARTRIDGE BACKER, this version of the program may not be backed up. A copy of the cartridge may be made by burning EPROMS and installing them on a cartridge board.

HOW TO MAKE A WORKING COPY: The easiest way to locate the program's protection is to use the COMPARE AND REPAIR function of the CARTRIDGE BACKER program. This function allows the user to narrow down the possible locations for the programs protection. It is possible to do this program without the use of the CARTRIDGE BACKER, so we will include both methods.

1. First we need a disk (RAM) version of the program. Use either OPTION 1 or 2 from the CARTRIDGE BACKER (CB) program (both will be identical); just be sure to obtain a printed report. For the sake of clarity, name your object file 'MB'. If you don't have the CB software, use the techniques outlined in the

Program Protection Manual (I or II). For review we will cover the technique here.

- a. Insert your cartridge and de-activate it (turn off the cartridge switch or turn off the proper switches on your expander board).
 - b. Turn on your computer, load HIMON and then SYS49152.
 - c. Activate your cartridge (by the switch or via your expander board).
 - d. Save out the cartridge memory with the following command: S "MB.OBJ",08,8000,C000
2. Now it is necessary to execute the RAM version of the program. If you are using the CB, simply load and execute the auto boot and allow the program to crash. Flip switch 2 ON and RESET the computer. Now flip switch 2 OFF and load the CB software. Follow the steps in OPTION 5 (COMPARE AND REPAIR) to identify the location that has been written to (\$85AF). Note that location \$8000 will be changed by the RESET routine of the computer and NOT by the program itself.

If you don't have the CB, follow these instructions:

- a. After you save the program out to disk (and you're still in the monitor), transfer the program into the RAM under the cartridge: T 8000 BFFF 8000
 - b. Now de-activate the cartridge via the switch(es). Turn off the BASIC ROM by changing location \$0001 from \$37 to \$36. Use the following command to execute the RAM copy: G 805A
- The program will crash. RESET your computer and re-enter your ML monitor.
- c. Turn off the BASIC ROM (change location \$0001 to \$36) and transfer the code down to \$2000: T 8000 BFFF 2000
 - d. Load the disk version of this program: L "MB.OBJ",08
 - e. Compare the two versions of the program: C 8000 BFFF 2000
 - f. The comparison will result in finding locations \$A000 and \$85AF as being different. In this case, location \$A000 was changed by the RESET routine and \$85AF was written to by program.
3. Now that we have established that location \$85AF is where the cartridge is writing to itself (thereby causing the program to crash), we can hunt for the code that writes to \$85AF. By the way, location \$85AF gets changed to the value of \$00. Since CB does not fix this program on its own, you can be reasonably sure that protection will be indirect indexing (STA (XX),Y). First thing to try is hunt for where the program sets up the pointers to do the indirect indexing. We can usually find this by doing a simple hunt for LDA's (A9) with the individual address bytes. Try the following hunts:

H 8000 BFFF A9 AF and H 8000 BFFF A9 85

The following locations will be returned: 80DE & 80E5. Take a look the programs ML code from \$80DE on. It will become apparent, during investigation, that this area of code (from \$80DE on) is setting up pointers at \$88 and \$89 (among other things). The pointers (\$88 & \$89) are pointing to location \$85AF. At \$8112 the program loads the accumulator with the value of \$00 (LDA #\$00). At \$811A the program transfers the value in 'A' (\$00) to the 'Y' (TAY), thereby also setting the 'Y' to \$00. Next (at \$811B & 811C) the program stores the value in the 'A' at location \$85AF with the command STA (\$88),Y. Remember that locations \$88 & \$89 point to \$85AF and that the value of 'Y' is \$00, so the final location will simply be \$85AF.

Keep in mind that a cartridge is ROM based and will not be changed if it tries to write to itself, whereas the same program residing in RAM can write to itself and cause the program to crash. All that is required to prevent the program from writing to itself is to change the STA (\$88),Y to NOP's. This is accomplished by changing the code at \$811B & \$811C to EA's. Just by changing these two bytes to NOP's the program will now run from RAM. If you had the CB software and printed a report of the possible locations where program protection may be hidden, you would find that the very first INDIR type of addressing listed is where the protection was located. This is why the CB software is so very valuable. For the few cartridges that CB will not backup, it is possible to use the report generated to find the protection.

4. If you are using the CB software all you need to do is save the modified version back to disk. You should replace the original version with the modified version: S "@0:MB.OBJ",08,8000,C000

At this point the CB version of this program will now load and execute properly from disk. It will require a little more work for those of you that don't have the CB.

If you don't have the CB it will be necessary to add a little bit of additional code to the program. This extra code will turn off the BASIC interpreter and then jump to the proper entry point of the program. For the sake of convenience we will add this code beginning at \$C000 (49152). This way when you load the program back in all that is required is to SYS49152 to execute this program. Use the 'M' command of your ML monitor to enter the code:

```
C000 A9 36 8D 01 00 6C 00 A0
C000 A9 36    LDA #$36    (DISASSEMBLY OF CODE)
C002 8D 01 00 STA $0001    TURN OFF BASIC
C005 6C 00 A0 JMP ($A000)    INDIRECT JUMP TO START OF PROGRAM
```

Save the modified version of this program back out to disk with the following command: S "@0:MB.OBJ",08,8000,C008

To execute this program just LOAD "MB.OBJ",8,1 then SYS 49152.

5. YOU'RE DONE

A final comment as to how this program auto-starts without the CBM80. As you recall this is a 16K cartridge that resides from \$8000 to \$BFFF. The cartridge will turn off the BASIC interpreter ROM and force the microprocessor to look to the cartridge port for the entire block of memory from \$8000 to \$BFFF.

During the normal initialization routine of the C-64 the computer will look for a CBM80 at \$8000. If it is found the entire initialization process will be turned over to the cartridge program. It must be pointed out that the initialization process is a quite lengthy and complicated process. If the CBM80 is NOT found the normal initialization will be continued by the computer. At the end of normal initialization the computer will perform an indirect jump into the BASIC interpreter - JMP (\$A000). This jump uses memory locations \$A000 and A001 as vectors. Normally these locations point to the REAL start of BASIC. When we install THIS cartridge the vectors at \$A000 have been changed to where the program begins. This is an easy way to let the computer perform all the initialization routines and leaves the programmer free to concentrate on writing his program (and not on initializing the computer).

The following was submitted by a newsletter subscriber:

The program is MICRO COOKBOOK, Copyright 1983 by Virtual Combinatics, Copyright 1983 by Commodore Business Machines.

TYPE OF PROTECTION: The program will check for two errors on the disk - an error 20 on both track 1, block 0 and track 35, block 1. This program uses an autoboot and machine language encryption.

HOW TO COPY: Use a copy program that will place the error 20's on the disk for you.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK

1. Format a disk and copy tracks 2-34 from the original to the copy disk using a suitable copy program.
2. Using a T/S editor, change the load address of the file "COOKBOOK" from \$0100 to \$C100. Do this by changing byte 03 on track 17 block 0 from \$01 to \$C1.
3. Load "COOKBOOK" now with LOAD "COOKBOOK",8,1. After loading, reset the computer.
4. Using LOMON, examine the code from \$C100-\$C23C. At \$C23A is the entry point to the main program. (This is used after the encrypted ML program called "COOKBOOK1" is loaded and decoded). Change the JMP \$C030 at \$C23A to a BRK (\$00). Also change the \$C3 at \$C153 to \$00. This will disable the cartridge

reset protection (CBM80) by changing the letter "C" .Then, using S"COOKBRK",08,C100,C257, save the modified autoboot to disk.

5. Load a T/S editor and change the load address of "COOKBRK" from \$C100 back to \$0100. Reset the computer.
6. Type LOAD "COOKBRK",8,1. The autoboot program will load and decrypt "COOKBOOK1", a SEQ file. Reload LOMON and save the now decoded "COOKBOOK1" with S "COOKBOOK1U",08,C000,C150
7. Write the following short loader program and save as "LOADERCOOK".

```
10 IF X=1 THEN 100
20 X=1
30 LOAD "COOKBOOK1U",8,1
100 SYS 49257
```

Starting the program with SYS 49257 (\$C069) will bypass the protection scheme.

8. To start the program just use LOAD "LOADERCOOK",8 and then RUN. The loader program could be also autobooted using a program like Superbooter. The now unneeded COOKBOOK, COOKBOOK1, and COOKBRK files can be scratched.

The program is MICROSOFT MULTIPLAN, Copyright Microsoft Corp., 1981,83.

TYPE OF PROTECTION: This program requires two checks for proper execution. The 1st check is for a good block at track 1 sector 2. If this block is good, the program will then check track 1 sector 1 for an error. The program does not check for a specific error, but only that one is present. It is a shame that such a good program uses a protection scheme that 'beats the disk drive to death'.

HOW TO COPY: Copy the disk with any good copy program and place an error 23 on the disk at 01/01. You may also use a nibble copy program and let it put the error on for you.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK.

- 1) Use BACKUP 228 to copy tracks 2-35, or use a good copy program that will not place errors on the disk.
- 2) Load and execute HIMON. Fill in memory from 0900-3600 with 00.
- 3) L "MP." 08 - This is the program that contains the error checking.
- 4) Using the M command change 3586 and 3587 to EA (NOP).
- 5) S "@0:MP." 08 0900 2600.

6) YOU'RE DONE.

This gets you up and running, but you haven't learned a thing. The Newsletter is an extension of the P.P.M., and is intended to keep you in touch with the developments in program protection. You will find that what you learn in the analysis of this program may be applied to many other programs currently on the market. If you've been asking where they're hiding the CMP#32's, this is the program for you. Since this program is a bit more complicated than some of the others we have investigated, we will take more of a tutorial approach to unprotecting it.

Let's examine the code in MP. with HIMON. The area of code that we are interested in is located from 3500 to 3608.

Indirect addressing - 3500-352C. Beginning at 3555, you will find references to #65, or memory locations beginning with \$65. This will send us off to look for the error checking routine at \$6500. Further investigation shows that there is nothing out there. This is the programmer's attempt to confuse us. When the section of code at 3500 is executed, it will change these references from 65 to 35, which is where the error routine resides. Once the code has been changed, following the logic of the code becomes easier. In order to see how this works, we will execute only this section of code. First transfer 3500-3600 to 4500. This will allow us to retrieve the original code for further study. Now we will change the code at 352E to a BRK. Using the M command change the AC too 00 (BRK). Inserting a break will allow us to execute this section of code and return control to us. We insert the BRK here so that we may discover the result of the indirect addressing. The code at 352E will check for the presence of a cartridge (machine language monitor). If a cartridge is detected, the program will fall through to the code at 353A, which erases the code and crashes the program. With the BRK inserted at 352E, we may now run this section of code with G 3500. Now when we disassemble the code from 3555 through 35DB, we will find our 65's changed to 35's. We discover a fully functional error checking routine.

Now we will find out what happened to our CMP#32's. We begin the routine a \$35B0. At \$35CA, we input a character from the channel and AND that value with #50F. After we return that value, we push it on the stack (PHA). We recall this value by pulling it off the stack (PLA). This value is used in the branch instructions at \$357E and \$3586. Keep in mind the BNE and BEQ instructions looks for a true or false situation. Let's see how this works. The first check we make is for a good block. Once we input the character from the channel (FFCF), we will return the following bit pattern.

```
3      0
0011 0000 - #30 - No error on this pass
0000 1111 - #50F - AND with #50F
-----
0000 0000 - #00 - This value is checked at 357E
```

When we AND a value with #50F, we eliminate the high nibble. Instead of checking for #30 or #32, we are now checking for #00 or #02. This makes the

code a little more difficult to spot and alter. Let's go back not to \$357E with our \$00. If any value other than \$00 is returned, the program will branch to \$353A, which erases the code and crashes the program. The DEC instruction sends us back to check the channel again. This time we will be checking for an error. If the error is there, the bit pattern will be as follows.

```
      3      2
0011 0010 - #$32 - An error is returned.
0000 1111 - #$0F - This will eliminate the high nibble.
-----
0000 0010 - #$02 - This value is checked at 3586
```

We are now sent to 3586. If we return with a 00, the BEQ will send us to the self-destruct code at \$353A. If the disk has an error at Track 1 Sector 1, we will return with the value of \$02 and the program will fall through the branch instruction. To get around the error, we change the code at \$3586 and \$3587 to NOP. The program will now fall through without errors on the disk.

That takes care of the protection. You are up and running and may now apply what you learned to other programs. When you are unsure of the purpose of a subroutine, do what we did here by inserting a BRK at a logical ending point in the code. Remember you cannot hurt the computer, but should have a backup of your test disk.

NOTE: Some companies may be updating their error routines on different releases of their programs. If what we suggest here does not work, the routine has probably been updated.

The program is **MILLIONAIRE V2.0**, copyright 1984 by **BLUE CHIP SOFTWARE**.

TYPE OF PROTECTION: This program uses a KEY (or "DONGLE") plugged into joystick port #1 for protection. The disk is not protected, so you can back it up with any copy program. However, the program will only run if the dongle is plugged in. This sounds like a good compromise between program protection and ease of backup, but what happens if the dongle gets lost or stops working? (both of these things have happened to us with dongles, although they're more dependable than disks). You still need to have a "backup" dongle, or better yet, a version of the program that doesn't check for the dongle. We haven't dealt with any dongle protection schemes in the Newsletter before, because they can be very complex and hardware-oriented. This scheme is simple enough for everyone to understand, and it's especially good for beginners since it's in BASIC (a similar technique could be used in machine language). As a bonus, we'll learn a little about how to read the C64's game paddles.

HOW TO MAKE A DONGLE: The easiest way to make a backup dongle is to use a pair of game paddles. Just plug the paddles into joystick port #1 and turn both paddle knobs all the way clockwise (only one paddle will actually be checked). That's all there is to it! Most dongles are much more complicated, but this one is quite

simple. Another way to make a dongle is to connect a wire from pin 5 to pin 7 of joystick port #1. Pin 5 is the top right pin on the port (as you look at it) and pin 7 is the second pin from the bottom left. A joystick extension cord or old joystick cable can be used to plug into the port, as long as the cable has connections to pins 5 and 7. If you want to match the characteristics of the original dongle more exactly (which isn't necessary), use a 2.2K ohm resistor between pins 5 and 7 instead of a direct connection. Actually, the program will run as long as the resistance is under about 8K ohms.

HOW TO MAKE THE PROGRAM RUN WITHOUT A DONGLE:

1. Make a copy of the disk with any copy program.
2. Load in the first program on the disk: LOAD "M",8. List line 15. Change the GOSUB 100 to GOSUB 110. Save the program back to the COPY disk with: SAVE"@0:M",8
3. Now load in HIMON and enter the monitor with SYS 49152. Fill memory with F 0801 3000 FF. Load the second program on the disk with: L"INITIAL.BAS",08
4. You need to change four bytes of memory (at \$105D, \$10E0, \$15ED and \$1E0F) from \$8B to \$8F. Display the first location using the memory display command M 105D. Type 8F over the first byte displayed (8B) and hit RETURN. Now change the other three locations the same way.
5. Save the program back to the copy disk with the command: S"@0:INITIAL.BAS",08,0801,2F47.
6. That's it - you're done! Reset the computer and run the program as usual. You might want to backup this version (it can be file copied if desired).

Let's follow along to see how to find the check for the dongle. Load in the first program on the disk ("M"). This is a BASIC program. Run the program WITHOUT the dongle to see what happens. Part of the title screen will be printed and then the computer will RESET itself without loading any files. This suggests that we should look for a SYS 64738 (or equivalent) in the program, since this command is the most common way to perform a RESET from BASIC. Reload "M" and list it (it's short). Line 33 says:

IF PEEK(A+L)<73 THEN SYS 64738 . That looks like the culprit! Change line 33 to read: 33 REM and run the program again (without the dongle). It still doesn't work! What's going on?

We need to look more closely at exactly when the program RESETS itself. In the program code, you can see that after printing the part of the screen that did appear, the program does a GOSUB 100 at line 15. Line 100 loads in a program called "VAR.BAS". But we know that the program didn't load any files before it crashed. What happened between printing the screen and doing the LOAD in line 100? Any time you think a BASIC program is doing something you can't see in the code, you should suspect HIDDEN LINES. These are lines of BASIC code that contain control characters such as DELETE (\$14), which prevent the lines from being

listed correctly on the screen (although the lines execute correctly). One way to read such lines is to use an ML monitor that can display memory in ASCII (such as HIMON's "I" command). Another way is to print out the program with an interface that converts control characters like \$14 to a more readable form like [DEL]. Most interfaces can do this.

If you print out this program, you'll see that line 100 actually starts out `IF PEEK(54298)>10 THEN SYS 64738`. The line continues with a REM statement containing some DELETE's and then the `LOAD"VAR.BAS"` statement that appears when we list the line on the screen. In fact, if you list line 100 by itself, you can see the hidden part flash briefly before it is deleted and replaced by the LOAD statement. What does the hidden check do? Well, location 54298 is used to read game paddles on the C64. The number in this location is related to the electrical resistance between certain pins on the joystick port. The higher the resistance, the higher the number that appears. A direct connection between the pins (no resistance) will produce a 0 value, while no connection (infinite resistance) will produce a 255. Plugging in the dongle produces a value of 3 in this location. Without the dongle, the value 255 appears. Thus when the dongle is NOT used, line 100 will RESET the computer. Notice how similar this is to disk protection schemes that return a particular value when the protection is found. The game paddles work as a dongle because they are VARIABLE RESISTORS; turning them clockwise reduces their resistance to 0.

To defeat this dongle check, all we have to do is change the GOSUB 100 to GOSUB 110, to skip the check entirely. Make this change and re-run the program. Sure enough, it passes the point where it stopped before! Unfortunately, it RESETS again a little later! To make a long story short, the file INITIAL.BAS has FOUR MORE dongle checks, three of which are hidden with DELETE's. You can find them with a hunt command from the monitor: `H 0800 3000 "54298"`. The checks are in lines 455, 505, 1295 and 2000 in the BASIC program (at the locations given in step 4). The procedure given in steps 3-5 demonstrates the best general method for changing these lines, since we don't change their length. Those changes simply replace the IF keyword with a REM, so the check is disabled. Note that when you save a BASIC program from a monitor, you must start at \$0801, not \$0800. Also, the ending address of a program (plus 1) can be obtained directly from locations \$00AE-AF after loading the program.

The program is **MONTEZUMA'S REVENGE (R)**, Copyright 1984, Parker Brothers.

TYPE OF PROTECTION: The disk is filled from TRACKS 1 to 13 and TRACKS 19 through 36 with ERROR 21'S. This results in three "HEAD BANGINGS".

HOW TO COPY: Copy with any good nibble copy program that will place the errors for you.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS:

1. An expander board is required for this program. A Cardco/5, Cartridge Backer or CSM Single Slot board will work. Power off the computer, insert the expander board and turn on the switch that grounds the EXROM line. This is switch 2 on the CB and CSM boards, and is the upper-right hand switch on the CARDco/5. Power on the computer.
2. Load and execute HIMON with SYS 49152. Provide a clean work space with F 8000 BFFF 99. With this accomplished, perform a RESET with G FCE2. The screen will now display 30719 BYTES FREE.
3. Load the program from the original disk with LOAD ":",8,1. When the program has finished loading and the game begins executing, perform a RESET with your reset switch. Re-enter the monitor with SYS 49152. With the M command, change the \$37 at \$0001 to a \$36. This will flip out the BASIC ROM.
4. Using the M command, change location \$8000 to \$09. This will replace the \$55, which was inserted as a result of the RESET routine. Refer to the PROGRAM PROTECTION MANUAL VOLUME II for a complete explanation of RESETS.
5. Save out the code from \$8000 to \$BFOC to a formatted disk with S "PBGAME",08,8000,BFCD.
6. To use this program LOAD "PBGAME",8,1. Activate with SYS 32777. This is the decimal equivalent for HEX \$8009, which is the warm start vector for this program.
7. YOU'RE DONE!

The program is MORSE UNIVERSITY (11.JUN.85), copyright 1985 by AEA Inc.

TYPE OF PROTECTION: This program is supplied on a cartridge (ROM). The program contains a routine which will cause it to crash if you try to run it in RAM memory (e.g. from a disk copy).

HOW TO COPY: The cartridge's ROM can be removed, read and copied to a new 8K EPROM (2764) by using the PROMENADE EPROM programmer. You'll also need a PC2 standard cartridge board set up for an 8K cartridge.

HOW TO MAKE A WORKING COPY ON DISK:

1. Insert the cartridge in the computer and turn it on. If you have an expansion board, plug it into the computer and insert the cartridge into it (always turn the cartridge power off before removing or inserting cartridges!). This cartridge is unusual in that it doesn't autostart on powerup. You should see the normal BASIC startup screen, except that it will show only 30719 bytes free.
2. Load in and run an ML monitor at \$C000 (49152).

3. If you DON'T have an expansion board, save the original version of the cartridge code to disk with: S "TEMP",08,8000,A000. Turn off the computer, remove the cartridge and turn the computer back on. Reload the monitor and run it. Load the original cartridge code from disk with L "TEMP",08.
4. If you DO have an expansion board, you can take a shortcut. Simply transfer the code from the cartridge to the underlying RAM using T 8000 9FFF 8000 . Now turn off the cartridge via the expansion board. The code should still be intact in RAM memory.
5. After performing either step 3 or 4 above, you are ready to modify the code. Display memory using M 843D. Change the first two bytes displayed (\$843D-843E) from \$91 0B to \$EA EA.
6. While you're at it, you may want to make another change. The minimum speed rate is set at 18 words per minute, which may be too fast for you. To change the minimum speed, you must alter the bytes at \$8708 and \$870C. Both bytes have the value \$12 (18) in the original program. Change BOTH bytes to whatever value you desire (e.g. \$05).
7. Save the program to disk with: S "MORSE UNIVERSITY",08,8000,A000
8. That's it - you're done. To run the program, use LOAD "MORSE>UNIV",8,1 and then SYS 32768.

This cartridge is a good Morse code tutor program (as you may have guessed already, some of us around here are avid ham radio operators, along with many of our readers). The program uses a very common method of cartridge protection, namely trying to write over part of itself. A ROM (cartridge) version of the program won't be altered by writing to itself, but a RAM version (loaded from disk) will. This way, a copy in RAM can be made to crash itself. The classic technique for detecting this type of protection is sometimes called "compare & repair". You compare a crashed version of the program with the original to see what bytes were altered, and then look for the code that altered them.

First, save the original program to disk, remove the cartridge and reload the program as we did in step 3. Now run the program with SYS 32768. It will crash - the screen will go blank and nothing will happen. Reset the computer. The crashed code will still be in memory. Load in an ML monitor at \$C000 and run it. Transfer the crashed code down in memory to the \$2000-3FFF area with T 8000 9FFF 2000. Reload the original program at \$8000. Now compare the original program with the crashed version using C 8000 9FFF 2000. This will show you all locations that have been altered. In this program there is only one, at \$80E5. Now we must find the program code that alters this byte.

First, hunt for a direct reference to address \$80E5 using H 8000 9FFF E5 80. No luck. Next, hunt for the \$E5 and \$80 bytes separately, and look for a place where they occur within a couple bytes of each other. Using H 8000 A000 E5 and H 8000 A000 80, you'll find that these bytes occur near each other at \$842B and \$842D. Examining this area reveals the "crash" code. First, a vector pointing to

\$80E5 is set up by storing an \$E5 and an \$80 byte, respectively, into locations \$000B and \$000C. A little later, the accumulator (A) and Y-register are set to \$00, and then a STA (\$0B),Y instruction is executed at \$843D. This instruction stores the accumulator into a location determined by adding Y to the value of the vector in \$000B-000C. Thus, a \$00 byte is stored into location \$80E5+\$00 (\$80E5), which is exactly what we saw happens. To prevent this, all we have to do is replace the STA instruction with NOP's (\$EA) as we did in step 5. Be sure you make this change to the ORIGINAL version of the program, not the crashed version. The program now runs from RAM.

The program is **MOVIE MAKER**, Copyright 1984, Interactive Picture Systems Inc.

TYPE OF PROTECTION: This program will check TRACK 2 for an error 23 and TRACK 3 for an error 27. That's right - TWO bangs for your buck.

HOW TO COPY: Copy the entire disk without errors and then put error 23's on all of TRACK 2 and error 27's on TRACK 3. Or use Omniclone, which will copy this program and place the errors for you.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

1. Make a copy of the original disk without errors (3 minute or file copy).
2. Load and execute LOMON with SYS>32768. Clean up the work space with F CC00 CFFF 99.
3. Load "MM0" from your original disk with L>"MM0",08. Make the following changes using the M command. These values are inserted by the program if it passes the protection scheme.

```
CC04      90
CEF8-F9   FF FF
CF50-52   54 48 45
CF56      01
CFDB-DE   01 01 01 01
```

Save this file to your copy disk with S "@0:MM",08,CC00,D000.

4. You can now load and run the program as usual (load "*" or "MM") EXCEPT for the following. After the main menu appears the first time, you should select the PLAY option. Once PLAY has loaded (it's short), you can return to the MAIN MENU and choose any other option as usual. If you want to skip the opening credits, start with LOAD>"MM0",8,1 and then SYS>52224. You should still choose PLAY the first time into the main menu.
5. YOU'RE DONE!

This program was done by the 'back-door' method. After starting the program normally from the original, we RESET the computer and examined memory. The "MMO" file contains the main menu and loads up at \$CC00-CCFF. By comparing this area of memory with the original file, we found the differences given in step 3 above. After saving the modified "MMO" file back to a copy disk, we began examining the program more closely.

We found that each option module has its own protection check code, which is encrypted using EOR (exclusive-or). On the original disk, once a module has passed its check the first time, a value is saved which tells that module not to check again. The other modules, however, will still check the protection the first time they are loaded. On a copy disk with the modified "MMO" file, we found that the PLAY option works. Not only that, but it disables the protection for all the other modules. As long as we always choose the PLAY option first, we won't have to unprotect each option separately. Sometimes a little experimenting can save a lot of work.

The program is THE CLONE MACHINE - MSD VERSION, copyright 1984 Micro-W. Distributing Inc.

TYPE OF PROTECTION: The disk contains an error 21 on track 4 and track 29.

HOW TO COPY: Use any copy program capable of producing these errors.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

Due to the heavy encryption utilized in this program, we will unprotect it once it comes into memory and passes the error checking routine. The boot program loads three modules. There is a section of code from \$0801 through \$4938 which will decrypt another section of code at \$C000.

1. Format a disk and file copy the programs called "START" and "CPYT1" to the formatted disk. We will use the original boot programs because their purpose is just to load in the main body of the program. These boots do not contain any protection routines. This will save us the trouble of constructing a boot program.
2. Load the program from the original disk. Once the title screen is up, RESET your computer.
3. Load the RESTORE program from your Program Protection Vol I disk and activate it with SYS 525 :CLR. Load LOMON and execute with SYS 32768.
4. Now that the program code has been decrypted and has passed the protection check, we may save out the files to the formatted disk using the original file names.

S "CLONE",08,0801,4938

S "B/M1",08,C000,C520

5. That's it - you're done!
6. The program may now be loaded in the same manner as the original disk. The only thing you're missing is the head banging at the end of the load!

While you still have your copy of LOMON in, do a comparison of the original and altered "B/M1" files. Load the original file and transfer it with T>C000>C51B>0C00. Once the file is transferred, load "B/M1" from the copy disk. Take a look at the end of each file with the M command. In the altered file, beginning at \$C4A3, you will find the copyright message and U1 command. In the original file, the code looks harmless enough. You can get a look at the end of the original file by scrolling through the code from \$10C3 through \$1113.

To gain better understanding of encryption and decryption, refer to the Program Protection Manual II.

The program is **THE MUSIC PROCESSOR**, Sight & Sound Music, Inc., Copyright 1984, M. Peter Engelbrite.

TYPE OF PROTECTION: This program will check for an error 29 on TRACK 11. If present, the program will run properly.

HOW TO COPY: Use any copy program capable of reproducing errors, such as OMNICLEONE.

HOW TO MAKE A WORKING COPY WITHOUT ANY ERRORS ON THE DISK:

Last month we provided a review of the earlier protection schemes. We hope it cleared up any questions you had regarding these procedures. This month we will concentrate on lifting a working copy of a program from memory after it has passed its error checking routine. Remember, this procedure often proves less time-consuming than tracing down the protection scheme.

Our first program requires that we utilize a new technique. This program will store code to the RAM under D PAGE (\$D000-DFFF). Capturing this code is similar to the technique used for programs stored under KERNAL RAM (\$E000-\$FFFF).

Before we get to the unprotection process, let's do a little review. There is a RAM section of memory under BASIC, D PAGE, and KERNAL ROM. In order to access this code you must change the address at \$0001 to flip out the ROM and expose the RAM. The normal value found at \$0001 is \$37. In order to access the codes under the ROM, we must change this value to the following:

BASIC RAM \$36

KERNAL RAM \$35 (BASIC ALSO)
D PAGE RAM \$34 (KERNAL AND BASIC ALSO)

To access the code under D Page, we will write a machine language routine to transfer the code from D PAGE (\$D000-\$DFFF) to \$2000-\$2FFF. We will construct this routine at \$1000.

Load and execute HIMON with SYS49152. Using M 1000 102A, will reveal the section of memory that we will use to store our program. Change the values to the following:

```
.:1000 78 A9 34 85 01 A9 00 85
.:1008 FB 85 FD A9 D0 85 FC A9
.:1010 20 85 FE A0 00 B1 FB 91
.:1018 FD C8 D0 F9 E6 FC E6 FE
.:1020 A5 FC C9 E0 D0 EF A9 37
.:1028 85 01 58 00 00 00 00 00
```

Check your program with the following disassembly.

```
1000 78    SEI               PREVENT IRQ INTERRUPTS
1001 A9 34 LDA #$34        VALUE REQUIRED TO ACCESS D PAGE RAM
1003 85 01 STA $01        STORE THE $34 AT ADDRESS $01
1005 A9 00 LDA #$00        LOAD THE LOW BYTE OF D PAGE AND 2000
1007 85 FB STA $FB        STORE D PAGE LOW BYTE HERE (00)
1009 85 FD STA $FD        LOW BYTE $2000 STORED HERE
100B A0 D0 LDA #$D0        SOURCE PAGE HIGH BYTE (D PAGE)
100D 85 FC STA $FC        STORE D PAGE HIGH BYTE HERE
100F A9 20 LDA #$20        DESTINATION PAGE HIGH BYTE
1011 85 FE STA $FE        STORE HERE
1013 A0 00 LDY #$00
1015 B1 FB LDA ($FB),Y     TRANSFER LOOP
1017 91 FD STA ($FD),Y
1019 C8        INY
101A D0 F9 BNE $1015     LOOP BACK TO $1015 UNTIL ALL CODE AT D
                          PAGE HAS BEEN TRANSFERRED
101C E6 FC INC $FC
101E E6 FE INC $FE
1020 A5 FC LDA $FC        CHECK $FC FOR END OF D PAGE
1022 C9 E0 CMP #$E0       ONCE AT $E000 END TRANSFER
1024 D0 EF BNE $1015
1026 A9 37 LDA #$37       SET LOCATION $01 TO NORMAL VALUE
1028 85 01 STA $01        STORE HERE
102A 58        CLI        ALLOW INTERRUPTS
102B 00        BRK
```

Once you are sure the program is correct, save it to a formatted disk with S "D PAGE TRANSFER",08,1000,102B.

The purpose of this program is to transfer a copy of the code under D PAGE to \$2000-\$2FFF. After the transfer, we will tack on a section of code that will transfer this original code back to D PAGE for our unprotected version. Let's get to it!

1. For those who have a copy of the PROTECTION MANUAL II load and run the program called "FILL'ER UP", which is included on your program disk. This program will fill memory with 99's, which will make it easier to locate the program code. The advantage of FILL'ER UP is that it will fill the memory under the ROMS. If you do not have "FILL'ER UP", LOAD and execute HIMON with SYS49152. Once executed, fill memory with F 0800 BFFF 99. Exit to BASIC with G FCE2 (SYS64738). Be sure to have a blank, formatted disk available for this program.
2. Load and execute the original program. Once the program gets to the menu screen RESET your computer.
3. Load and execute HIMON. We will now go after the D PAGE code. Clean-up the work space with F 1000 4000 99.
4. Load the transfer program from your formatted disk with
L "D PAGE TRANSFER",08.
5. Execute the program with G 1000. Using the I command from \$2000-\$2FFF, will reveal the code that occupies the RAM under D PAGE. If you used "FILL'ER UP" you will find that the actual code begins at \$2800 (\$D800).
6. We will now tack on a routine to transfer the code at \$2000 to D page. We will reverse the process we just used to transfer D PAGE to \$2000. We will even use the same program. Once we change the source and target pages, this program will reverse the process. T 1000 102B 3000 will make a copy of the original transfer program to \$3000. Use the D command at \$3000 to check to see if your transfer worked. We will now alter the code at \$3000 to change our source and target pages. This will require that we change D PAGE references to \$2000 and that we change the comparison for the end of our code at \$3000. Note: we can't just load code directly into D page from the disk. Unlike BASIC and KERNAL roms, a store to D page will go to the I/O devices rather than RAM (see the chapter on the '6510 AND THE PLA' PPM VOL II).
7. Use the M command to make the following changes:

ADDRESS	ORIGINAL VALUE	NEW VALUE
M 300C	D0	20
M 3010	20	D0
M 3023	E0	30

Along with changing the target and source pages, we will add a RESET instruction to the new transfer program. Use the M command to add the following code at \$302B.

M 302B 4C E2 FC 00 00 00 00

Once the D PAGE code is loaded and executed with SYS12288, this section of code will return us to BASIC through the RESET vector (JMP \$FCE2 = SYS64738).

Save out the D PAGE code and the transfer program using
S "MP D PAGE",08,2000,302E.

8. We will now go after the main body of the program. Clean-up the work space again with F 0800 C000 99. Exit to BASIC with G FCE2.
9. Load and run the original program. Once the menu screen is up, RESET your computer.
10. Load and execute HIMON. Use M 0001 to change the 37 to a 36. This will flip out BASIC and allow us access to the code underneath. The program extends from \$0800 through \$A995. Save out the code with S "MP MAIN",08,0800,A995.
11. Now for the entry point. Using the D command, disassemble the code at \$7D1A (SYS32026). This section of code will load the program called "START.UP" (C000-CFFF) which is the menu screen. That's all there is to it.
12. File copy all programs beginning with "START.UP" to your formatted disk. To execute this program use the following procedure:

LOAD "MP D PAGE",8,1	(RETURN)
SYS12288	(RETURN)
LOAD "MP MAIN",8,1	(RETURN)
SYS32026	(RETURN)

13. YOU'RE DONE!

The following was submitted by a newsletter subscriber:

The program is **THE MUSIC SHOP**, copyright 1985 by Broderbund.

TYPE OF PROTECTION: This program executes a custom DOS routine that looks for nonstandard data on track 2 and elsewhere.

HOW TO COPY: Some of the newer parameter copiers can copy this program.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

1. Copy the disk using a copier that does not put errors on the copy disk.
2. Load and run a track/sector editor.

3. Read in track 17, sector 10 from the copy disk. Change the first six bytes in the sector to the following and write the sector back out to the COPY disk.

\$A9 FF 8D FF 01 60

4. That's it - you're done.

To examine the disk protection scheme, load and run the program. After it is in memory, RESET the computer. Examine the memory area \$6450-6473. A BLOCK-EXECUTE (B-E) command is executed here for track 17, sector 10. With the original disk, this custom DOS routine will place an \$FF in the disk drive RAM at \$01FF if it finds the nonstandard bytes it is looking for. The program will next read (with an M-R MEMORY-READ command) the disk drive RAM at \$01FF and check for an \$FF. If found the program executes normally. If not, the program goes into an endless loop. The changes to the copy disk in step 3 above place the expected value in disk drive RAM, bypassing the original custom DOS routine. The DOS routine is changed to:

```
LDA #$FF
STA $01FF
RTS
```

The program is **THE NEWSROOM**, Copyright 1985 by Springboard Software, Inc.

TYPE OF PROTECTION: Track 35 is heavily protected. The same protection is used on the master program disk and the Clipart data disks.

HOW TO COPY: None of the standard nibblers would copy the program or data disks. A parameter-type copier may be able to duplicate these disks.

HOW TO MAKE A WORKING COPY WITHOUT PROTECTION ON THE DISK:

1. Make a copy of the master program disk with a copy program which does not put errors on the disk.
2. Load and execute a track and sector editor.
3. Make the following changes to the COPY disk:

TRK/SEC	BYTES	ORIGINAL	CHANGE TO
26/5	\$2B-2D	\$20 0E 3E	\$EA EA EA
27/3	B0-B2	20 CB 5A	EA EA EA

4. Copy the Clipart data disks with a copier that does not put errors on the disk. No changes to the Clipart disks are necessary.
5. That's it - you're done.

This program is very difficult to copy. To uncover the protection method, the code was traced through much of the boot process. The boot program loads the file "XXX.O", which sets up the title screen from the file LOGO. Then it loads the file "B.NEWS.O". This file loads in a number of other files. Rather than trace the whole start-up process, we looked through some of the files that "B.NEWS.O" loads to see what might turn up. Sure enough, some very interesting memory-read (M-R), memory-write (M-W) and memory-execute (M-E) commands were found in the file "MN.O". These are stored backwards in memory at \$3EA6-D7 (i.e. W-M, R-M etc.).

To find where these commands were used, we searched for references to \$3EA6 using the monitor command H A6 3E. The commands are executed by a subroutine at \$3E71, located just before the commands themselves. Different commands may be sent to the drive by specifying different values in the accumulator (A) before calling the subroutine. Next we searched for where the \$3E71 routine is called, and found it is used by subroutines at \$3E41 and \$3E0E. The subroutine at \$3E41 is called only from the \$3E0E routine.

The \$3E0E subroutine is the protection routine. It is relatively short and straight-forward to trace. First the values \$21 and \$10 are written to drive memory at \$0006-07, using a M-W command. These locations are used to specify the track and sector (respectively) for job queue command #0. In this case, the decimal value for the track/sector is 33/16. Next, the value \$80 is written to location \$0000. This puts a sector read command into job #0. Job #0 uses buffer #0 at \$0300-FF. Thus block 33/16 is read into drive memory at \$0300. The read job stores a status byte in drive memory \$0000 when it is finished. The protection routine waits until the job is completed by reading location \$0000 and waiting for it to change from \$80 (read job code).

If the job status indicates an error in the read attempt, the routine initializes the drive and tries again. The drive initialization is done directly by executing a subroutine at \$D042 in the drive's DOS. On the other hand, if the read job was successful, the code read into \$0300 from block 33/16 is executed (in drive memory). The code at \$0300 checks the protection on track 35 and stores a value in drive memory at \$0006.

We don't have to worry about what kind of protection is used on track 35 because all that matters is whether the correct protection value is stored at \$0006. The protection routine in the computer simply reads location \$0006 and checks to see if it is an \$FF. If the value is not correct, the routine starts all over again at \$3E0E. This causes the drive to hang up on track 35 on a copy disk. On the other hand, if the value at \$0006 is correct, then the start-up process continues and the program runs normally. The protection value \$FF is not used in any other way by the program.

So how do we defeat the protection check?. One way would be to run the routine from 33/16 on a copy disk and see what value is returned at \$0006. The check in the computer could be altered so it only accepts the value from the copy disk rather than the \$FF from the original. As we'll see, there are reasons to look for another way to disable the check. Perhaps the \$3E0E protection routine

can just be skipped altogether. By searching through the MN.O file, we found that the \$3E0E routine is called only at \$3C21. The first change in step 3 above changes the bytes at \$3C21-23 to NOPs (\$EA) to skip the \$3E0E routine. After this change, the backup copy now loads perfectly.

Loading in the program is not the whole story, however. One of the options from the master menu allows you to load in pictures from the Clipart data disks. These data disks have the same protection on track 35 as the master program disk. The protection is checked every time you load in a picture, but a DIFFERENT routine is used than was used on the program disk. Once again, we started looking through the remaining files to find the second routine. We found some M-W, M-R and M-E commands in a file called "PH.O". In fact, both the disk commands and the subroutines are IDENTICAL to the ones in the "MN.O" file, except they are relocated in memory. Instead of being at \$3E0E, the protection routine is at \$5ACB. The disk commands are at \$5B63. The \$5ACB routine is called at \$57E1.

Once more, we could have altered the \$5ACB routine to accept the value from a data disk copy, rather than an original. However, then we wouldn't be able to use the program to read both original and backup data disks. This situation prompted the experimenting that proved the protection routine could be skipped entirely. This way, the protection is never checked, and either original data disks or backup copies will work with the program. All we have to do is replace the call to \$5ACB (located at \$57E1) with NOPs, as we did before. This is accomplished by the second change in step 3 above. The backup copy now works just as well as the original!

NEWSROOM UPDATE:

There is a new version of Newsroom on the market. The protection is exactly the same as explained in the January '86 Newsletter, except that the protection routines are in different sectors. On the new version we've seen, the changes we gave should be made to blocks 19/1 and 20/5, respectively, at the same relative byte positions. Alternately, load the files "MN.O" and "PH.O" with an ML monitor. Search for the original bytes we gave, change them to NOP's and replace the files.

When run, many programs will alter their code to check the error channel. They will then store these values within the program. If the correct value is not found, the program will not execute properly. These programs are a bit more difficult to unprotect, but with a little effort the result is the same. Programs of this type become easier as we gain experience working with them.

The program is **NIGHT MISSION PINBALL**, Copyright, Sublogic Corporation., 1982

TYPE OF PROTECTION: The program is stored on the disk in user files. The program checks track 03 for an error 21.

HOW TO COPY: Use a good copy program and place an error 21 on Track 03. You may also use a nibble copy program and let it put the errors on for you.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK.

PROBLEMS:

- 1). The program is stored in user files making it difficult to examine the code.
- 2). The program will check the error channel after a program option is selected.
- 3). A portion of the program is stored under basic.
- 4). An entry point must be found.
- 5). We must write a small section of code to flip out basic and then jump to the proper entry point.

SOLUTIONS:

- 1). Load and execute HIMON.
- 2). Using the M command at 0001 change the 37 to a 36. This will flip out the basic interpreter. Fill the memory from 0800-BFFF with 00. Change location 01 back to a 37.
- 3). Using the X command, exit to basic and load and run the program from the original disk.
- 4). When the option menu appears, choose regular mode, but do not play the game.
- 5). Reset your computer.
- 6). Activate HIMON with SYS 49152.
- 7). Using the M command at 0001, change the 37 to a 36. This will allow us to save out the code stored under basic.
- 8). Disassemble the code at 093A. This is where the program will check the error channel. Since we have already run this section of code and have the values we need stored in memory, we will eliminate this section of code. This will keep the program from overwriting the values we need for proper execution. Using the M command at 093A, change the A9 to a 60 (RTS).
- 9). Now we have to write the boot to flip out basic and jump to the entry point. We store this code in a section of memory that is not being used by the program. Using the M command, we find that 0A00 is available. Change the first 7 bytes at 0A00 to the following: A9 36 85 01 4C 61 08. When we disassemble the code, we find the following:

LDA #36 - Load the accumulator with the value of 36.
STA \$01 - Store this value at 01 - This will flip out basic.
JMP \$0861 - This will take us back into the program.

10). Using the I command, locate the ending address of the program. You will find that point at B098.

11). S "NIGHTSYS2560",08,0800,B098

12). YOU'RE DONE.

When you load this program in, you must SYS to 2560. This is the decimal equivalent for 0A00, which is where we stored our program to flip out basic. It's always a good idea to save a SYS location in the program name in case you forget the location.

The only real trick to this program is finding the entry point. This is really a matter of experimentation. We searched for a spot that would put us into the menu screen.

The program is New York Times Crossword Puzzles Vol 1, Number 1. Copyright 1984 by Softie, INC.

TYPE OF PROTECTION:

The program checks for an error 23 on track 3 sector 2 and an error 27 on track 3 sector 16. The main program is compiled.

HOW TO COPY:

This program can be copied with any good copier capable of reproducing errors 23 and 27.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS:

- 1) Make a copy of the disk without errors (4 min. copy or equivalent will do).
- 2) Load a track and sector editor and read in track 15 sector 3.
- 3) Change bytes 75 through DD to 3A (an ASCII colon).
- 4) Save the sector back to disk.

EXPLANATION:

I was originally going to SNAPSHOT the program, but since it checks for errors before loading any files as well as during the initial load, it would not work. I examined the main program on the disk and found it to be compiled with INSTA-SPEED. Searching through each sector of the file I

finally came across a BLOCK-READ command for track 2 sector 15 then track 3 sector 16. I noticed that colons were used as line separators and wrote colons over the section of code checking for the errors. After a few trials and errors I found out exactly which bytes to replace. The program can now be copied by anything and loads without 'head banging'!

The program is OMNI CALC, COPYRIGHT 1983 ISA SOFTWARE.

TYPE OF PROTECTION: This program checks track 1 sector 00 for an error 20, HEADER NOT FOUND. The program uses a BASIC loader to load in the main ML program, it then checks the error channel and does a SYS to the start of the ML program (SYS0512 = \$1524). The BASIC program does not do any comparison, it just checks the error channel and inputs a character (X). The value of the X is stored in the computers memory in the normal variable storage area (immediately after BASIC). The first thing the ML program does (at \$1524) is reads the area of memory where the variable X is stored (LDA \$09CD). The value of X is loaded into the accumulator and then stored as a part of the program (STA \$7DBB). The program uses the value of X as an instruction in the ML code. So, if the error is not the proper type the program will not execute properly.

HOW TO COPY: Copy the disk with any good copy program and place error 20 on the disk at 01/00. You may also use a nibble copy program and let it put the errors on for you.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK.

The technique used to break this program is similar to the techniques described on pages 14-18 and pages 50-60 of the Program Protection Manual.

- 1) Use BACKUP 228 to copy the disk, or use any good copy program that will not place any errors on the copy disk.
- 2) Load and execute the original program. RESET the computer. Load and execute the HIMON.
- 3) Disassemble the code starting at \$1524 - \$1540 (\$1524 is the start of the ML program). D1524 1540
- 4) The program will first load a value from \$0924. Disassemble the code at \$0924 and see what value the program is looking for (\$20).
- 5) Fill the memory of the computer with 00's from 0800 to 9FFF.
F 0800 9FFF 00
- 6) Load in the ML program called CALC64. L "CALC64",08 Find the beginning and ending address of the program. Start: \$0A00 End: \$7E00

- 7) Change the code at \$1524 as follows:
1524 A9 20 LDA#\$20
1526 EA NOP
 - 8) Save the program back out to the copy disk: S "@0:CALC64",08,0A00,7E00
 - 9) YOU'RE DONE....TRY YOUR LUCK ON THE OTHER PROGRAM ON THE DISK (PLOT64).
-

The program is **ON-COURT TENNIS**, Copyright 1984, Gamestar.

TYPE OF PROTECTION: This program places an error 20 on tracks 1-3. The program will check two bad blocks and one good one. If the proper values are returned, the boot program will load the main file and execute properly.

HOW TO COPY: Make a copy of the disk and place an error 20 on tracks 1-3, or use a copy program that will place the errors for you.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

- 1). File copy the main program called "FILE" to a formatted disk.
- 2). When we take a look at the loader program (TENNIS) through a track and sector editor (track 19/0), we find the loading address to be \$0100. This is the **STACK**. The stack is located in memory from \$0100 to \$01FF. The programmer is filling the stack with the address for the actual area of execution, which is \$0203. This is determined by the fact that we find a repeated pattern of 02's in the disassembled boot. When the loading address (\$0202) is pulled off the stack, it will be placed in the program counter and then incremented by one. This will make the starting address \$0203. This is a nice form of protection. At \$0203, we will find a jump (JMP) to 0800, which indicates that something will be happening there. The boot code is encrypted on the disk and will be altered when it is brought into memory. This will make it impossible to alter the code on the disk. Since we cannot alter this program on the disk, we will have to go about it another way.
- 3). Load the program from the original disk with, LOAD " :* ",8,1. Watch the disk drive error light. When the light begins to blink, reset your computer. This gives the boot program enough time to decode and expose the error checking routine.
- 4). Load and execute LOMON with SYS 32768. Disassemble the code beginning at \$0800. As you can see, we now have a working loader. At \$08C5, we find a CMP#\$32. This is the check for the bad block. Later in the code you will find a CMP#\$30, which is the check for the good block. Using the M command, change \$08C6 to a 30.
- 5). We will now save the altered code to our formatted disk with, S

"TENNIS",08,0800,0A41.

- 6). The only thing left to do is to find an entry point. Another good place to try an entry point is at a jump instruction (JMP). When we disassemble the code again at \$0800, we find the first JMP is to \$0908. With the disk containing our two files in the drive, we will try a G \$0908. That's all there is to it.
- 7). To utilize this program, LOAD "TENNIS",8,1 and execute with SYS 2312 (HEX 0908).
- 8). YOU'RE DONE.

You will find auto boots described in far greater detail in the PROGRAM PROTECTION MANUAL VOLUME II. As stated, the stack is located in memory from \$0100 to \$01FF. When an address is pulled off the stack it is placed in the program counter. The program counter is a 16-bit register, which contains the address of the next command to be executed. Once the program counter gets its address from memory, it is incremented by one, pointing to the next memory location to execute. Through this boot, \$0202 will be placed in the program counter. When incremented, this address becomes \$0203. At this location, we find a jump (JMP) to \$0800. It is here that we find the program protection scheme.

Always check the starting address for an auto boot program, through a track and sector editor. This can save you a lot of time. If you do not have a track and sector editor that contains a monitor feature, just change the starting address of the program to an area of memory that may be examined through one of your monitors. Try it with this program.

Go to track 19/0 on the original disk and change the 01 to a C1. The program will now locate to \$C100 and may be examined through LOMON.

We chose programs for the NEWSLETTER that illustrate the current trends in program protection. Keep track of each new technique you learn and apply them to other protected programs. If one technique doesn't work, try another.

If you find you have a different version of the program than the one described, don't throw in the towel. The basic technique is there. You may just need a different entry point. Get out your memory map and analyze the code in that area. All you can lose is a little time, but you may gain a great deal of knowledge.

The program is OXFORD PASCAL, Copyright 1984 O.C.S.S & D. Goodman.

TYPE OF PROTECTION: The disk contains error 23's on track 1, sector 0 and track 13, sector 9. Only the error at 13/9 is checked.

HOW TO COPY: Use any copy program which can copy error 23's.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

- 1) Copy the disk with any copy program which doesn't create errors.
- 2) Load a track/sector editor and run it.
- 3) Read in sector 18/1. Bytes 64-95 (\$40-5F) should be all \$00's now.
Change bytes 66-72 (\$42-48) to \$82 0D 05 42 4F 4F 54.
Change bytes 73-84 (\$49-54) to \$A0's (shifted spaces).
Change byte 94 (\$5E) to \$0E.
Write the altered sector back to disk.
Note: the bytes in the sector are numbered starting at 0, not 1.
- 4) Read in sector 13/5.
Change byte 4 (\$04) from \$04 to \$08.
Change byte 131 (\$83) from \$32 to \$30.
Change byte 138 (\$8A) from \$33 to \$30.
Write the altered sector back to disk.
- 5) Exit from the T/S editor. List the disk. The third file should be a new program 14 blocks long called BOOT.
- 6) Scratch the files PASCAL and LD from the disk. Validate the disk using the DOS command "V0". List the disk. You should have 134 blocks free.
- 7) You're done!

The first program on the disk, PASCAL, is a BASIC program which just loads in the file LD. LD is an autoboot routine (using the stack) which loads at \$0100-\$18FF and starts executing at \$1770. LD then loads 14 blocks from disk into the BASIC area at \$0801-1521. This 14-block program starts at sector 13/5 and is linked together like a regular program file. In step 3 above we created a directory entry for this program (in a conveniently empty slot) and named it BOOT. Refer to PPM>Vol>I for details on how directory entries are set up.

Let's take a look at how LD can load in the BOOT program without a directory entry. Remember, each sector of a normal file starts off with a two-byte link to the next sector of the file. The normal LOAD process gets the first link from the program's directory entry, and simply follows the links from there. LD starts by disabling the NMI vector (RESTORE key), initializing the disk (IO) and opening a random access channel (#). Then it initializes some key locations: \$45-46 holds the address to begin loading at (\$0801), \$47-48 holds the first link (\$000D 05 = 13/5) and \$49 holds the number of bytes to read from each sector, not counting the link bytes (\$FE). Next, the main routine calls a subroutine at \$182E-64. That routine calls on another routine at \$1865-8A, which converts a hex byte into its two-byte ASCII equivalent and puts the result into \$62-63. In this case, the routine is called twice to convert the T/S link to ASCII, since a UI command

requires its T/S info in ASCII. The ASCII T/S info is then written into a 'blank' U1 command at \$17AB-B6. The U1 command is sent to the drive, which gets the specified block. After any U1 command, the disk sends an error (or OK) message, so a routine at \$18CA-DC is called to read in the message without doing anything with it (just to 'shut the drive up').

Finally, we return to the main routine with the first block ready to be read into computer memory. The first two bytes contain the next T/S link, so a subroutine at \$18ED-FA is called which simply opens up the data channel (2) and reads in one byte. This byte (track #) is stored at \$47 and then the routine is called again to get the next byte (sector #), which is stored at \$48. This saves the T/S link for the next U1 command. Next, a routine at \$1825-2D is called which checks to see if this is the last sector of the file. Since it isn't, we'll postpone looking at that routine until a little later. In fact, this is the very first block of the file, so the next two bytes are the load address. The main routine is going to load the file into memory based on its own pointer at \$45-46 (set to \$0801 earlier), so the load address bytes are simply read in by the \$18ED routine and ignored. Location \$000D will be used to count the number of bytes read from the sector so far (not including link bytes), so this counter is set to \$02 to account for the load address bytes.

At long last we are ready to read in the actual program code from the first sector. The routine at \$18ED is used to get the bytes one at a time. The subroutine at \$18E2 stores each byte in the location pointed to by the pointer at \$45-46, which is then incremented. Back in the main routine the byte counter in \$000D is incremented too. Once the number of bytes specified by \$49 has been read in, we are ready to get the next sector, assuming there is one. The routine at \$182E is called to convert the T/S link to ASCII, send out the U1 command and skip over the error (OK) message. Then the first two bytes of the new sector (next T/S link) are saved in \$47-48 and checked to see if this is the last sector. Since it isn't the last sector, we simply set the number of bytes read so far (\$49) to \$00 and jump back to the code which reads in the rest of the sector.

The process described in the last paragraph is repeated over and over until the last sector is encountered. The LD program detects this the same way the normal LOAD routine does: the first byte of the T/S link is \$00 (there is no track 0). The second byte of the link then specifies where in this block the file ends (since the block may not be full). The routine at \$1825 transfers this byte to location \$49 (number of bytes to be read). Then the rest of the block is read in, up to the specified byte. The error (OK) message is skipped, and the files are closed. The real BOOT program has now been read in and is ready to be executed.

The BOOT program is executed with JMP \$080F (=2063). The author has conveniently preceded the code at \$080F with a BASIC program consisting of SYS (2063), which means we can start the program with RUN. The program checks sector 13/9 for an error 23 in a completely straightforward manner. Step 4 above changes the check for error 23 into a check for no error (00) by changing a CMP #\$32 and CMP #\$33 to CMP #30's as usual. Also in step 4, we change the default load address of the BOOT program from \$0401 to \$0801. This is not absolutely

necessary, but it avoids problems if you should accidentally load BOOT using ",8,1" rather than just ",8". Once we have put BOOT into the directory, we no longer need the PASCAL and LD files. We scratch them and then validate the disk to correct the BAM (which indicated 0 blocks free originally). The disk can now be file copied or have other programs added to it. To start PASCAL, simply load BOOT and RUN.

The program is **PANZER BRIGADIER**, copyright 1985 by SSI.

TYPE OF PROTECTION: The disk has an error 23 on track 33, sector 5.

HOW TO COPY: Any nibble copier will copy this disk.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

1. Make a copy of the disk with no errors on it.
2. The disk is software write-protected. Run the UNWRITE PROTECT program from the June 85 Newsletter on the COPY disk. Many utility packages such as DI-SECTOR also have a similar utility.
3. Using a track and sector editor, read in track 17, sector 2. Change bytes \$B4-B5 (180-181) from \$32>33 to \$30>30. Save the sector back to the COPY disk. That's it - you're done!

The error checking is done in a BASIC program called "B". You can load "B" from BASIC and examine it. Step 3 changes the check for an error 23 to check for no error (00). To be able to list the disk, change the disk name in bytes \$90-9F (144-159) of sector 18/0 with your T/S editor.

The following was submitted by a newsletter subscriber:

The program is **PFS FILE Version A.00**. Copyright 1984 by Software Publishing Corp.

TYPE OF PROTECTION: The program checks for an error 23 on track 2, sector 15 and an error 27 on track 3, sector 16.

HOW TO COPY: Use any copy program that can copy error 23's and 27's.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

1. Copy the disk with any copy program that doesn't create errors.
2. Load a track/sector editor and run it.

3. Read in track 13, sector 4. Change bytes \$72-74 from \$20 B4 FF to \$4C 38 84. Change byte \$AC from \$20 to \$60. Write the revised code back to the same track and sector.
4. Read in track 11, sector 17. This is an unused area of the original program code. Change the code at bytes \$36-5F as shown below and save the new code back to the same track and sector:

```

$30:  .. .. .. .. A9 17 85 0A 84 0B AE 61 84 D0
$40:  16 A9 1B 85 0A A2 20 8E E6 68 8E AC 68 A2 B4 8E
$50:  AD 68 A2 FF 8E AE 68 A2 00 CE 61 84 4C DB 68 01

```

5. The program is now unprotected and can be file copied.

ANALYSIS OF UNPROTECTION SCHEME:

The changes on track 13, sector 4 cause the program to skip the actual error reading and storage by inserting a JMP \$8438 at \$68AC-\$68AE in the computer memory. An RTS is also inserted at \$68E6 in memory. The code written to track 11, sector 17 resides in the computer memory at \$8438-\$8461. This code stores the expected error values in memory and restores the original program code. The new code is disassembled below.

\$8438 A9 17	LDA #\$17	\$844C 8E AC 68	STX \$68AC
\$843A 85 0A	STA \$0A	\$844F A2 B4	LDX #\$B4
\$843C 84 0B	STY \$0B	\$8451 8E AD 68	STX \$68AD
\$843E AE 61 84	LDX \$8461	\$8454 A2 FF	LDX #\$FF
\$8441 D0 16	BNE \$8459	\$8456 8E AE 68	STX \$68AE
\$8443 A9 1B	LDA #\$1B	\$8459 A2 00	LDX #\$00
\$8445 85 0A	STA \$0A	\$845B CE 61 84	DEC \$8461
\$8447 A2 20	LDX #\$20	\$845E 4C DB 68	JMP \$68DB
\$8449 8E E6 68	STX \$68E6	\$8461 01 00	ORA (\$00,X)

Analysis of the new code shows a storage of #\$17 at \$0A the first time through. This is the value supplied by the original error reading routine located in memory at \$68A5-\$691B and \$6FC0-\$6FF2. A branch byte at \$8461 of the new code is set to \$00 after the first error value is stored. The second time through a value of #\$1B is stored at \$0A. The new code from \$8447-\$8456 restores the original values to the program code. This effectively defeats the error protection by merely supplying the values the program expects to find in location \$0A. The new code required a vacant spot in the program which was conveniently available.

The program is PHARACH'S CURSE, COPYRIGHT 1983 SYNAPSE SOFTWARE

TYPE OF PROTECTION: This program is stored on the disk in user files. Once in memory, the program checks for errors on track 4 sector 11 and track 17 sector 01 for error 23 CHECKSUM ERROR IN DATA. If these errors are present, the program

will run. Tracks 19 through 35 contain error 21, but these are not checked by the program.

HOW TO COPY: Copy the disk with any good copy program and place the errors on the disk at 04/11 and 17/01. You may also use a nibble copy program and let it put errors on for you.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK.

The technique used on this program is similar to the techniques described on pages 76 and 77 of the Program Protection Manual.

- 1) LOAD and RUN the original program. During the load, the screen will go through a series of color changes. When the screen stops on Orange, the program will read the error channel. Immediately after, the screen will turn to Cyan. RESET the computer at this point (page 34 of P.P.M.). If you are unsure of what color you are looking for, use these POKES in direct mode, POKE 53281,3:POKE 53280,3. This will turn your screen to cyan. If you get into the option screen, your RESET was too late and your computer will not RESET properly. If you RESET in the proper spot, you will have a normal RESET.
- 2) LOAD and execute HIMON. Go the location \$0001 and change the 37 to 36. This will flip out the BASIC interpreter, allowing you to save out the code stored there. This code is necessary for the program to function properly.
- 3) Save the code from 0800 to C000. (EX. S "PHARO",08,0800,C000)
- 4) Load "PHARO",8,1 RETURN : SYS3756 RETURN

The SYS 3756 will take you to the code at HEX OEAC. This ML code changes the screen to CYAN, as you saw in our POKE test. Locating the proper entry point is the most difficult part of the process. One of the first places to start is a HUNT for a STA at \$D021 AND D020. You will find this explained on page 78 of the P.P.M.

```
OEAC A9 03 LDA #$03
OEAE 8D 20 DO STA $D020 (DECIMAL 53280 BORDER COLOR)
OEB1 8D 21 DO STA $D021 (DECIMAL 53281 SCREEN COLOR)
```

Remember that after the program checked the disk and found the error, the screen turned CYAN. All you had to do was to find the ML code that turned the screen CYAN (\$OEAC) and use this location as the new entry point for the program (SYS 3756).

- 5) YOU'RE DONE.

The program is **PIECE OF CAKE**, Copyright 1984, COUNTERPOINT SOFTWARE.

TYPE OF PROTECTION: This program will check TRACK 2 SECTOR 2 for an error 23. If present, the program will run properly.

HOW TO COPY: Use any good copy program capable of placing errors on the destination disk.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

The protection scheme used for this program is similar to the title. Disappearing line numbers is the only form of protection used. When you LOAD the boot program called "PIECE OF CAKE" and attempt to LIST, you will notice that the lines begin to disappear at 40 and continue through 52. The best way to examine this program is to list it to the printer, but it may also be examined through LOMON.

1. Make a copy of the program without errors on the disk, or file copy the programs to another disk.
2. LOAD "PIECE OF CAKE",8. List the program.
3. Line 40 will require a bit of repair, because part of the line is erased when it is listed.
4. Line 40 should read as follows: ONAGOTO73,75,77,99,100,200,202 Don't forget to type RETURN.
5. We will now delete the check for the bad block. Move to the bottom of the screen and type the following:

47 (RETURN)
48 (RETURN)
49 (RETURN)
50 (RETURN)
51 (RETURN)
52 (RETURN)
6. You may now save the program to your copy disk with SAVE "@0:PIECE OF CAKE",8.

The program is POKER SAM, by Peggy and Jerry White. Copyright 1983 by DON'T ASK SOFTWARE.

TYPE OF PROTECTION: The program checks track 1, sector 1 for an error 23. The program is stored in encrypted random blocks.

HOW TO COPY: Use any good copy program which copies bad block errors.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS:

- 1.Run the original program. Once the music starts, reset the computer.

2. Use `LOAD"HIMON*",8,1` to load HIMON (keep the monitor name under 7 characters. Switch off the BASIC ROM by changing location \$01 to \$36.
3. Make the following changes:

LOC	FROM	TO
\$1058	\$9E	\$8F
10A5	9E	8F
4. Save out the main section of memory using `S "POKER 2",08,0800,C000`
5. Reset the computer. Rerun the original program. Reset the computer again when the music starts.
6. Load in LOMON. Save the rest of memory using `S "POKER 3",08,C000,D000`
7. Reset the computer again. Type in the following BASIC program:


```

10 IF A=0 THEN A=1 :LOAD "POKER 3",8,1
20 POKE 43,1 :POKE 44,16 :REM BASIC PROG START = $1000
30 POKE 45,28 :POKE 46,76 :REM " " " END = $4C1C
40 CLR :LOAD "POKER 2",8,1
      
```
8. Save the BASIC boot program: `SAVE "POKER SAM",8`
9. That's it - you're done!

The program is stored on disk in random blocks. The "U1" command is used to directly load each block of memory. The segments of memory are also encrypted. Each byte is exclusive-or'ed (using the EOR instruction) with the value \$FF to decrypt it. This has the effect of simply flipping each bit, i.e. changing all 0's to 1's and vice versa.

The boot program checks track 1, sector 1 for an error 23. Once the error is found, the program has completely passed its protection check. It goes on and loads in most of the rest of the program. After the program starts (when the music plays), we can reset the computer without disturbing memory. If we look around in memory, we see some interesting things. The section of memory from \$1000 to \$4C1C is a BASIC program. This is the main program, although it has some machine language subroutines to handle the speech, etc. When the main BASIC program is RUN, one of the first things it does is load in two sections of code using an ML subroutine at \$52972 (\$CEEC). Since we already have all of memory loaded in, we have to disable these SYS calls. In step 3 above, we change the SYS's to REM's.

Now all we have to do is save memory. We have to do this in two sections as detailed above. Finally, we write a BASIC boot which loads in the first section, sets the BASIC pointers to the correct locations for the main BASIC game program, and then loads the rest of memory.

The program is **PROFESSIONAL TOUR GOLF**, Strategic Simulations, Inc., by Henry L. Richbourn, Copyright 1983.

TYPE OF PROTECTION: This program uses bad blocks as its protection scheme. The program will check for an error 23 on TRACK 2, SECTOR 15 and an error 27 on TRACK 3, SECTOR 16. Your disk drive will get a real "KICK" out of this program.

HOW TO COPY: Use any copy program capable of duplicating errors or make a copy of the disk and place the errors yourself.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

Although this program is not as intricate as the others, it does illustrate an important concept.

The program that does the error checking is "GS". Since the program is in BASIC, it may be examined easily. We find the BLOCK-READ commands in LINE 181 and LINE 183. LINES 182 and 183 will act on these values. In the first check in LINE 182, we are checking for a value < than 23. If the value returned is not 23 then "OK" becomes "NO". We find the same type of instruction in line 183. Here a value other than 27 results in a "NO". LINE 184 will act on these comparisons. If a "NO" was returned through the comparisons, the program will be sent to LINE 185. This section of code will crash the program. If the comparisons do not reveal a "NO", the program will be sent to LINE 186 and will execute normally.

It would seem that the most effective approach would be to delete the error checking routine, but there is a hitch. Since this program contains a machine language section, we must keep the program length the same, so that the machine language section will reside in its proper place in memory. We discussed this problem last month, but it bears repeating. With that in mind, let's unprotect the program. One last comment about this program. The program would not successfully operate by just changing the '23' or '27' to '00's. Evidently the program checks for the proper error number to be present in the BASIC portion of the program.

1. Copy the original disk with any program that will not produce errors on the copy.
2. LOAD "GS",8
3. Change the "NO" in LINE 182 to a "HO". Now list LINE 183 and change the "NO" to "HO".
4. Save the altered program with SAVE"@0:GS",8.
5. You may now load and execute the program in the normal manner. LINE 184 is checking for a "NO" not a "HO", so the program will be sent to LINE 186 and will execute properly.
6. YOU'RE DONE!

A program of this type seems very simple, but can give you fits if you are unaware of the concepts explained here.

The program is **PROMAL**, Copyright 1984 by Softspoken, Inc.

TYPE OF PROTECTION: The disk contains an error 23 on track 18,18.

HOW TO COPY: Copy with any copy program which can create error 23's.

HOW TO MAKE A WORKING COPY WITH NO ERRORS ON THE DISK:

1. Make a copy with any copy program which doesn't put errors on the disk.
2. Load and run a monitor at \$8000 or above.
3. Load in the first program on the disk with L "PROMAL",08.
4. Change location \$09B5 from \$D0 to \$F0. Change \$0AA8 from \$69 to \$A9.
5. Save the altered file out to the disk with :
S "@0:PROMAL",08,0801,46DB.
6. That's it - you're done.

This one can be done by working backwards from the message printed when the copy fails the protection test. This message is located at \$0AD2. First, we might try hunting for this address in lo-byte/hi-byte order. This doesn't work. Next, we hunt for all the \$D2 bytes and \$0A bytes separately and try to find a case where both bytes occur near one another. Aha! At \$09CE we find some code which prints the message and then goes into an endless loop. Backtracking a bit, we find a check at \$09B5 that sends us there. The first change in step 4 changes this check. But the program still doesn't run. This section of code looks like it starts at \$0943, so we search for where \$0943 is called. This is at \$0A83. A little farther down, we see some code that EOR's together \$0900-FF and adds \$77 to the result. The final result comes out to \$77 by coincidence (or design?). This is used as the location to JMP to. The second part of step 4 makes sure \$77 is obtained with the altered code, by loading A with \$77 rather than adding A to \$77. It works!

The program is **THE QUEST**, Copyright Penguin Software, Inc., 1984

TYPE OF PROTECTION: This program requires an error 23 on track 2 and an error 27 on track 3. If the errors are present, the program will execute properly.

HOW TO COPY: Copy the disk with any good copy program and place an error 23 on

track 2 and an error 27 on track 3. You may also use a nibble copy program and let it put the errors on for you.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK.

PROBLEMS:

- 1). The program that contains the protection (AUTO) is an auto boot, making the code difficult to examine. Page 52 of the P.P.M will give you some hints on auto load programs.
- 2). The program loads the accumulator (LDA) with the values it will later compare (CMP).

SOLUTIONS:

- 1). Make a copy of the original disk without errors.
- 2). So that we may examine the code without the program taking over control of the computer, we will load the program in the area of memory normally used for BASIC programs. Load "AUTO",8. Do not use ,8,1. Now execute HIMON and disassemble the code at 089E. Keep in mind that we are examining a program that would normally reside at 0300. At 089E, we find LDA #\$33 and STA at \$03EF. If we disassemble 08EF (03EF), we find a CMP #\$37. At 08AF, we find LDA #\$37 followed by a STA at \$03EF. Looking further, we find a CMP #\$32 at 08E8 and CMP #\$37 at 08EF. The program will check the channel four times. We must change the LDA's and CMP's to 30 to unprotect this program.
- 3). The easiest way to alter this program is to make the corrections on the disk. Let's get to the program through a track and sector editor. At 18/01, we find that the program we are interested in (AUTO) is stored at 17/03. If we go to 17/03, we find the first block of the file. In ASCII mode, we find the B-R for tracks two and three. If we go to HEX mode, we will find our A9's and C9's. Let's remember that what we see stored on the disk is similar to the M command in a monitor. LDA is stored as A9 and CMP is stored as C9.
- 4). Look for A933. This will load the accumulator (LDA) with 33. Change the 33 to a 30. It is not necessary to change the A9. Now look for the A937 and change the 37 to a 30. That takes care of the loads, now let's change the compares.
- 5). Find the C932 and change the 32 to a 30. With that accomplished, find the C937 and change the 37 to a 30.
- 6). With the four changes made, use your write function to write the changes to the disk.
- 7). YOU'RE DONE.

There are track and sector editors on the market that include an assembler function. You may find a program of this type helpful with auto load programs. If

you find it difficult to interpret the code found on the disk, a program of this type is what you need. Refer to the New Products Section for a review of such a program. You may also use the U1 and U2 programs included on the P.P.M. disk. These programs will require a few extra steps, but they will do the job. Page 79 of the P.P.M. contains the information necessary to use these programs.

The program is **QUESTRON**, Copyright 1984 by Charles Dougherty. Game structure and style used under license of Richard Garriot.

TYPE OF PROTECTION: There may be different versions of this program on the market. The version we have has errors on tracks 23-35 (side 1) and tracks 31 & 35 (side 2). The program only checks for an error 29 on track 35, sector 0 (side 1).

HOW TO COPY: Any nibbler should be able to copy the version we have. Other versions may require a parameter copier. Fast Hack'em includes a parameter file for Questron which DOESN'T work with the version we have.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

1. Copy BOTH sides of the disk using a copy program which does not put errors on the copy.
2. Load and run a track/sector editor.
3. Insert the COPY disk and read in track 7, sector 5 (side 1). Examine the bytes listed below to see if they contain the original values given. If so, change the bytes to the new values given and save the sector back to the disk.

BYTES	ORIGINAL	CHANGE TO
\$3C-3F	\$C9 32 D0 50	\$A9 32 38 EA
4C	C9	A9
4E	D0	F0

4. If the original values given above DON'T match your disk, DON'T make any changes to sector 7/5 - you probably have a different version of the program. In this case ONLY, read in track 17, sector 7 and try the following changes (same as Fast Hack'em's parameters).

BYTE	CHANGE TO
\$2A	\$BB
C1	80

5. That's it - you're done!

The boot program (QUESTRON) is in BASIC and appears to load in the file "ST". However, some of the BASIC statements are hidden by delete characters (\$14 bytes - see PPM Vol. I). The program actually sets the beginning of BASIC to \$6E00 and loads in another BASIC program called "M". The first thing "M" does is call a subroutine at line 10000, which pokes a small ML routine into the \$C000 area and executes it. This routine loads an ML program called "COMBINED.ML" into the \$C000 area and then returns to "M". Using the "COMBINED.ML" routine, "M" loads in several files next, including one called "SONG" which resides in the cassette buffer at \$0334-03DC. This is the protection routine and is executed with a SYS S statement. The variable S was set to 827 (\$033B) via a hidden line back in the boot. The protection routine does a block-read (B-R) of block 35/0 and checks for an error 29. Both digits of the error number are checked and the first one ("2") is stored at \$0311. In step 3 above, we place the same values in the accumulator and carry flag that they would have with the original disk. It works!

The program is REVERSAL, COPYRIGHT 1983 BY HAYDEN BOOK COMPANY, INC.

TYPE OF PROTECTION: The main program is stored on the disk in a program file and is written in ML (you can tell that it is written in ML by the speed of operation). This program is loaded into memory by a loader program that checks for the proper errors on the disk and executes the main program if the proper errors are found. The loader program is written in basic.

HOW TO COPY: Simply copy the disk with any good copy program and place the errors on the disk at the proper location. Or use one of the newer nibbler copy programs and let it put the errors on for you.

HOW TO MAKE A WORKING COPY WITH OUT ANY ERRORS ON THE DISK.

The technique used here is similar to the techniques described on pages 14-18 of the Program Protection Manual.

- 1) Copy the original disk with BACKUP 228 or any other good copy program that does not place any errors on the destination disk.
- 2) Load and list the loader program. Your can't do it can you? The program has a graphics character at the end of line 10 (see page 15 of the Program Protection Manual). To remove the graphics character all you have to do is to move the cursor up to line 10 and press RETURN.
- 3) List the loader program. Here is the entire program protection scheme! Look for and find the comparisons (IF E < 23 THEN and IF E < 27 THEN).
- 4) Change the value of the comparisons to 00. Be careful not to lengthen or shorten the code. (Some programs not only check for bad blocks, they also check the length of the loader program). When you copied the disk the errors were not placed on the destination disk. Now, when the error channel is read

the message 00,OK,00,00 will be returned from the disk drive. This indicates that there was not any error on the disk.

- 5) Save the code back to the copy disk. Use the save with replace option or scratch the file from the disk then save the modified version.
- 6) YOU'RE DONE.

The program is **ROCKY'S BOOTS**, Copyright Learning Company, 1984, C-64 version by Reliable Micro Systems.

TYPE OF PROTECTION: This disk contains an error 23 on TRACK 1/1. Error 20's are placed on 14/20 and 16/20.

HOW TO COPY: Use OMNICLEONE to copy this disk.

HOW TO MAKE A WORKING COPY OF THE DISK WITHOUT ERRORS:

- 1). Make a copy the original disk without errors.
- 2). When we investigate the disk directory through a track and sector editor, we find that the disk name contains illegal characters which will prevent us from obtaining a proper listing of the directory. Let's change the disk name. Use your track and sector editor at TRACK 18/00 to change the disk name. We will change the first two bytes before the disk name from 0D93 to 524F. We must also add a proper ID# to this disk. Change bytes A2, A3, and A4 to 524FA0 respectively. With these changes, the directory will list properly.
- 3). Load LOMON and activate it with SYS 32768. Using the F command, fill memory from 0800 through 7FFF with 99. We will also fill C000-CFFF with 99. This will help us locate the code necessary for the program. We will now perform a warm reset with G FCE2 (SYS64738). We will capture this program by lifting the code from memory. We will allow the program to store the codes necessary for proper execution and save us the trouble of tracing through the protection scheme. Load the program in the normal manner from the original disk.
- 4). Once the program has loaded and the main title screen is up reset your computer with your reset button. You will be returned to the normal blue screen.
- 5). Reactivate LOMON with SYS32768. The program has now past its protection scheme and may be saved out in two small sections. Using the I command, scroll through memory beginning at 0800. The first section of code that we encounter is at \$0803. Using the D command, we may determine that this is part of the protection scheme. Returning to the I command, we find that the first section of meaningful code begins at \$4800 and ends at \$5D20. Save this section of code to your copy disk with S "ROCKY1",08,4800,5D18.

- 6). To access the next section of code we must flip out BASIC. Using the M command at 0001, change the 37 to a 36. We will now return to the I command to continue our hunt through memory. The next section of code is stored from \$B508-CD73. Save this section with S "ROCKY2",08,B508,CD73. You will find various patterns throughout memory, but experimentation will show that they are not necessary proper program execution.
- 7). We now have the code that we need, but must find an entry point to the program. This is the tricky part. Let's go back to the section of code at \$0803. When we begin a search for an entry point, we find that a JMP instruction can be a good place to start. We will find a JMP instruction within the program protection code at \$082B. The JMP is to \$C800. Let's give it a try with G C800. As you can see, we are returned to the title screen. Our entry point worked!
- 8). To utilize this program, you must load in ROCKY1,8,1 and ROCKY2,8,1. To activate the program use SYS51200. This is the decimal equivalent for C800, which is our entry point. You may wish to write a boot program that will load the program parts and SYS to the entry point.
- 9). YOU'RE DONE.

Finding an entry point can be a frustrating process, but there are some guidelines that you can follow. Entry points are discussed in detail in the PROGRAM PROTECTION MANUAL II.

The program is **SAFEGUARD 64 DEMO**, Copyright 1985 Glenco Engineering Inc.

TYPE OF PROTECTION: This program uses error 22's on track 35, sectors 8-15. These errors are created in a special way which makes them difficult to reproduce. The program also uses an identification number which is stored on track 35 sector 7.

HOW TO COPY: We managed to make a working copy using Di-Sector's 3 minute copy program and placing the error 22's on track 35 with their format editor. This method was not totally reliable, however.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS:

- 1) Copy the original disk with any copy program which does not copy the errors.
- 2) Load in HIMON or LOMON and execute it. From the monitor, type
L "DEMOSUPP1",08.
- 3) Change location \$406A from a \$CE to an \$FF: type M 406A to display the location, type FF over the value displayed and hit return. Actually, almost any value other than \$CE would work here!
- 4) Save out the altered program with S "@0:DEMOSUPP1",08,3500,4280

5) You're done - just change one byte and you've bypassed their protection!

SAFEGUARD 64 is a program protection system. They supply blank disks with their errors already on them (\$2.85 each) and a routine to check for those errors. You then copy your software onto the disks. In your software, you call their error-checking routine and examine the values returned to see if the disk is an original. They also have an autostart option that will check for their protection and then automatically load your program. This option also allows you to encrypt your software.

The protection system is in two stages. First and foremost are the error 22's on track 35. If they are present, a 5 is placed in memory at \$4029. Then the program retrieves a five-digit ID number from sector 35/7 and places it in memory at \$CFFB-CFFF. The ID no. may be used in decrypting your program. The protection routine itself is located at \$4000-425F.

Although the protection system is not very effective, we can still learn something from it. Error 22 is a 'data block not found' error. Like most bad block errors, there are many ways to cause an error 22. In this case, rather than just alter the data block identifier byte (the simplest way), they have written some special data there. They either use altered bit density or illegal GCR codes such as \$00 bytes. This confuses the read circuitry of the drive. The net effect is that when you try to read the block, you get different results each time. This interesting technique may be the basis for several other tough-to-copy programs we've seen which use an error 22. The same technique could probably be used to cause other types of errors too.

To check for a bad block, the program simply reads the block into the computer's memory twice, once at \$CD00 and again at \$CE00, and then counts the number of bytes that are different in the two versions. Actually, it only counts up to a maximum of 5 bytes difference over the 8 possible bad blocks. This 'bad byte' count is kept in location \$4029, so that a 5 is placed there when the disk is an original. The method given above for disabling the protection check works very simply. Instead of comparing the first version in memory (at \$CD00-CDFF) with the second (at \$CE00-CEFF), we've switched it so the first version is compared with part of the KERNAL. Naturally, they come up with more than four bytes difference!

The program covered above is a demo of the protection system. In actual practice, the protection check routine is located at \$CAB0-CD20. The value 5 is returned at location \$CAB0, and the routine can be defeated by changing location \$CAFC from a \$C6 to an \$FF. The ID no. is returned at the same locations, \$CFFB-CFFF.

As software protection goes, this system is very ineffective and easy to defeat. It can be copied using the simple procedure given above. Although it may take several tries, you CAN get a working copy. Of course, why bother when disabling the protection check is so easy? The code that checks for the bad blocks is short and fairly simple. No attempt is made to hide any of the code through encryption or undocumented opcodes. Although they don't protect your

program, at least they don't destroy your drive either: Glenco disables the head rattling before checking for the bad block. In fact, they use the same technique that we published in the October newsletter.

It's important to realize that you would not know how ineffective this system is unless you investigated it as we did. This brings up a good point. When you let someone else protect your software, you are putting yourself in their hands. If you can't protect the program yourself, how can you properly evaluate their protection method? You can't. You just have to trust them. About the only thing you have to go by is their reputation and experience. I think that's why we've had so many requests from people to evaluate their protection system or protect their software for them. Because of the number of these requests and the lack of any good place to refer them, we have developed a number of protection systems.

Why more than one system? One reason is that different products require different levels of protection to get the best value for your program protection dollar. But the most important reason is one that is often overlooked: if a company offers one system to all its clients, that's putting a lot of eggs in one basket. If one egg gets 'cracked', chances are they all will. Our advice is: if you need your program protected, don't use an 'off-the-shelf' protection scheme. They're just not worth the risk. Bad protection is worse than none at all, since you've wasted your money and been lulled into a feeling of false security. These systems are not worth having at any price. To top it off, the prices they are asking would be out of line even if their systems were good. We have yet to see ANY of these systems that were worth the money.

In conclusion, the problem of finding a good protection system has no easy answer. Sometimes the best solution is to design one yourself, if you are capable of doing that. Most people aren't, since even good programmers don't usually make good 'protectors'. The other alternative is to use a commercial system. But remember what we have learned from the SAFEGUARD system: if it sounds too good to be true, it probably is.

The program is SARGON III, COPYRIGHT 1984 HAYDEN SOFTWARE.

TYPE OF PROTECTION: This program checks track 2 sector 15 and track 3 and sector 16 for an error 23 CHECKSUM ERROR IN DATA. You will also find an error 23 in track 1, but the program does not check this track. If the errors are present, the program will run properly.

HOW TO COPY: Copy the disk with any good copy program and place error on 23 on the disk at 02/15 and 03/16. You may also use a nibble copy program and let it put the errors on for you.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK.

The technique used to break this program is similar to the techniques described on pages 50-60 of the Program Protection Manual.

- 1) Use BACKUP 228 to copy tracks 4-35, or use any good copy program that will not place errors on the disk.
- 2) The program that does the error checking is called Copyright 1984. This program is located in memory from C000 to C3A0. Finding the starting address is explained in the P.P.M. on page 35.
- 3) Load and execute LOMON. Fill the memory from C000 to C3A0 with 00. This will help you identify the ending address of the program.
- 4) Using the I command you can locate the program's call for a Block-Read (B-R) at track 02/15 and 03/16.
- 5) Using the D command, disassemble the memory at C395. At this location, you will find a CMP #\$30. If the blocks being checked do not contain the errors, the program will crash. If the errors are present, the program will execute properly. If we change the comparison to CMP #\$32, we are now telling the program to crash if it finds an error. Since the disk will not contain errors, the program will fall through and execute properly.
- 6) S "@:Copyright 1984",08,C000,C3A0.
- 7) YOU'RE DONE.

The program is **THE SCROLLS OF ABANDON (R)**, COPYRIGHT 1984 BY ACCESS SOFTWARE INC.

TYPE OF PROTECTION: This program uses half-tracking on the original disk, making it very difficult to unprotect. It loads in an \$AA from the half-tracked section of the disk and will exclusive-or (EOR) this with a section of code at \$C000.

HOW TO MAKE A COPY OF THE PROGRAM:

- 1). You must first make a "CLONE" of the original disk. Use either Backup 228 or any other good copy program. Be sure to back-up the ENTIRE disk.
- 2). Load your track and sector editor so that we may alter the disk directory of the "clone" disk. At 18/01 you will find that the pointer to the next track and sector points to 18/01. This places the directory in an endless loop. We will place a stop indicator in these two bytes. Change byte 00 from 12 to 00. Change byte 01 from 01 to FF.
- 3). Load and execute LOMON with SYS 32768. We will provide a clean work space with F 0800 7FFF 00. We will now transfer BASIC ROM to RAM, with T A000 BFFF A000. With this accomplished, reset your computer with G FCE2 (SYS64738).

- 4). When the computer is reset it places a 55 at \$8000. If we follow the reset routine through, we find that this will effect the first byte at \$A000. Since no CBM 80 is encountered in the \$FD02 routine, it will execute the ROM test at \$FD50. During this test a \$55 is stored at \$A000. We must restore this byte with a poke from BASIC. In direct mode, type POKE 40960,148 (\$A000 - \$94) and press RETURN. The first two bytes at \$A000 contain BASIC's cold start vector. This address should be \$E394.
- 5). When the program has finished error checking, the loader uses the BASIC Re-link routine before executing a RUN from the input buffer. We can prevent this from occurring by replacing the RTS with a BRK instruction within the BASIC Relink routine. In direct mode type POKE 42335,0 and press RETURN (REMEMBER - We are now altering the copy of BASIC that we transferred in step 3). After placing the BRK in BASIC, we will switch from BASIC ROM to BASIC RAM with POKE 1,54.
- 6). Insert the original disk and type: Load ":",8,1. After the program loads, the computer will respond with READY. This was caused by the BRK we inserted.
- 7). Re-activate the monitor with SYS32768. Insert the backup disk and save the code from \$0801 to 08EA, with S "SCROLLS",08,0801,08EA.
- 8). YOU'RE DONE.
- 9). To run the program, LOAD "SCROLLS",8 and RUN.

These two program's bring up some interesting concepts that should be examined in more detail. Both routines require the restoration of the first byte at \$8000 or \$A000. These bytes are altered through the reset routine and RAM test. Let's examine the reset routine with concentration on the \$FD50 routine (RAM test).

RESET ROUTINE

FCE2 LDX #\$FF	
FCE4 SEI	PREVENTS IRQ (NO INTERRUPTS ALLOWED)
FCE5 TXS	SET THE STACK POINTER - MUST BE DONE ON THE 6502/6510
FCE6 CLD	CLEAR DECIMAL FLAG - NECESSARY FOR HEXADECIMAL PROGRAMMING ON 6502/6510
FCE7 JSR \$FD02	CHECK ON ROM IN \$8000 - CHECKS FOR CBM80 (CARTRIDGE)
FCEA BNE \$FCEF	
FCEC JMP (\$8000)	JUMP TO CARTRIDGE START
FCEF STX \$D016	SET VIC CHIP
FCF2 JSR \$FDA3	INITIALIZE IO
FCF5 JSR \$FD50	*EXPLAINED ON PAGE 7*
FCF8 JSR \$FD15	SET HARDWARE & IO VECTORS (0300+)
FCFB JSR \$FF5B	INITIALIZE VIDEO CONTROLLER
FCFE CLI	ALLOWS IRQ (INTERRUPTS TO OCCUR)
FCFF JMP (\$A000)	TO BASIC COLD-START

Let's break down the \$FD50 routine. This is the routine that initializes the work area and places the \$55 at \$8000 and \$A000. The commented code is as follows:

INITIALIZE WORK AREA - RAM TEST

```

FD50 LDA #$00      THIS SECTION OF CODE
FD52 TAY          WILL RESET ZERO PAGE
FD53 STA $0002,Y   PAGE 2 AND PAGE 3
FD56 STA $0200,Y
FD59 STA $0300,Y
FD5C INY
FD5D BNE $FD53
FD5F LDX #$3C      INITIALIZE CASSETTE BUFFER
FD61 LDY #$03
FD63 STX $B2
FD65 STY $B3
FD67 TAY          THIS IS THE SECTION THAT WILL
FD68 LDA #$03      CHECK FOR THE END OF RAM
FD6A STA $C2       THE COMPUTER WILL CHECK FOR THE
FD6C INC $C2       END OF RAM BY TAKING A VALUE OUT
FD6E LDA ($C1),Y   OF MEMORY (LDA). IT WILL THEN
FD70 TAX          PRESERVE THAT VALUE (TAX)
FD71 LDA #$55      FROM THERE IT WILL ATTEMPT TO
FD73 STA ($C1),Y   STORE (STA) A $55
FD75 CMP ($C1),Y   AND THEN COMPARE IT (CMP) THAT VALUE.
FD77 BNE $FD88     IF IT COMES BACK WITH A VALUE OTHER
                  THAN $55, THEN IT HAS ENCOUNTERED ROM. IF THE $55 CAN BE
                  PLACED, THE COMPUTER ASSUMES RAM.
                  CONTINUE CHECKING WITH $AA
FD79 ROL
FD7A STA ($C1),Y
FD7C CMP ($C1),Y
FD7E BNE $FD88
FD80 TXA          RESTORE MEMORY TO IT'S PRIOR
FD81 STA ($C1),Y   VALUE
FD83 INY
FD84 BNE $FD6E     CALCULATE FREE RAM
FD86 BEQ $FD6C
FD88 TYA
FD89 TAX
FD8A LDY $C2
FD8C CLC
FD8D JSR $FE2D
FD90 LDA #$08
FD92 STA $0282     SET START OF BASIC
FD95 LDA #$04
FD97 STA $0288     SET SCREEN
FD9A RTS

```

It should be noted that with the EXROM line on, the computer sees 52K of RAM from memory location \$0000 to \$CFFF. It does not find ROM at \$8000 and as a result cannot find the CBM80. This is how we defeat the auto-start in EASY SCRIPT (R).

The program is **SIDEWAYS**, Copyright 1985 Funk Software, distributed by Timeworks, Inc.

TYPE OF PROTECTION: Track 10 has a non-standard GCR format. Custom DOS routines are located on track 35 - sectors 0, 3 and 13.

HOW TO COPY: The nonstandard GCR is difficult to copy with copy programs. The Fast Hack'em parameter copier can make a working archival copy of this disk.

HOW TO MAKE A WORKING COPY WITH NO ERRORS:

1. You must use the "SWINSTALL" program to configure the **SIDEWAYS** program for your combination of printer, interface, etc. Thanks to their protection scheme, you have to alter your **ORIGINAL** disk to do this!
2. Load the **SIDEWAYS** program. When the main screen appears, **RESET** your computer.
3. Load in **HIMON** or other monitor at \$C000. Save out the program to a formatted disk with: S "SIDEWAYS 33096",08,0801,9600
4. You might want to copy the sample print file "SAMPLE.PRf" onto your backup disk with a file copy program. You don't need any other files from the original disk.
5. To run the program, type the following:

LOAD "SIDEWAYS 33096",8
SYS 33096
6. If you want to write an autoboot for this program, you should do it in machine language rather than BASIC. The BASIC area at \$0800 appears to be used by the program. SYS 33096 corresponds to JMP \$8148 in ML.
7. That's it - you're done!

Believe it or not, the protection scheme used by this program requires that you write onto your original disk in order to configure the program for your system. The **INSTALL** program contains different modules for each printer and interface (in fact, it's longer than **SIDEWAYS** itself). You select your particular combination and **INSTALL** writes the corresponding program code directly onto the original disk. And not just one sector - more than two tracks! Tracks 30 and 31 are completely rewritten, and parts of tracks 28 and 29 are too.

Once you've configured your original disk, the hard part is over. You can make an archival backup of the INSTALLED program for everyday use, by following the simple procedure above. We tested this procedure on several different combinations of printers and interfaces.

The program is SKY TRAVEL, copyright 1984 Commodore & 1984 Deltron.

TYPE OF PROTECTION: The program checks track 18, sector 18 for an error 22.

HOW TO COPY: Any nibbler will copy this program.

HOW TO MAKE A WORKING COPY WITH NO ERRORS ON THE DISK:

1. Copy the disk with any copier which does not put errors on the disk.
2. Load in a track/sector editor. Insert the COPY disk in the drive.
3. Read in block 35/5. This is the first block of the last file in the directory. The filename looks like a long underline (CHR\$(164)'s).
4. Change the following bytes. The bytes in the sector are numbered starting at \$00:

BYTE	FROM	TO
\$23	\$AC	\$AE
2A	AC	AE

5. Save the altered block back to the copy disk.
6. This disk has the annoying feature that the boot program (PLANBOOT) is not the first file on the disk. While you've got the T/S editor booted up, why not fix this? You just have to switch the directory entry for PLANBOOT with the first directory entry (DATA.TABLE). To do this, read in block 18/1 and change the following bytes:

BYTES	CHANGE TO
\$02-0E	\$82 17 08 50 4C 41 4E 42 4F 4F 54 A0 A0
1E	02
42-4E	81 0F 00 44 41 54 41 2E 54 41 42 4C 45
5E	21

This program from Commodore bumps the drive not once, not twice, but THREE times!! There's no excuse for this, especially from Commodore. Fortunately, they made the protection check very simple and easy to find. The boot loads in at \$0102 and uses the stack to autostart itself at \$0203. The code at \$0203-34 opens up the file mentioned in step 3 above and jumps to \$012C. The code at \$012C-4E decrypts the bytes of the file and stores them starting at \$C000. Each byte is decrypted by first EXCLUSIVE OR'ing it with the value \$FF (EOR #\$FF), which flips all the 1 bits to 0 and vice versa. Then the value \$21 is subtracted and the

result is stored. When the whole file has been decrypted, the program jumps to the decrypted code at \$C000.

The code at \$C000-16 opens up a random access file (OPEN2,8,2,"#") and jumps to \$COA7. The code there sets up 18/18 as the track & sector for a U1 command. The subroutine at \$C043 sends the U1 command. The subroutine at \$C01A-44 checks the error channel for an error 22 and crashes the program if it isn't found. The bytes of the error number are each checked using a CMP #\$32. We need to change each \$32 to a \$30 (00, OK) to disable the check. Since the file is encrypted on the disk, we have to figure out what an encrypted \$30 would be. To encrypt a byte, just do the opposite of decrypting it. First we ADD \$21 + \$30 to get \$51 (0101 0001), and then we flip all the bits to get \$AE (1010 1110). A similar process will show that the original \$32's were encrypted as \$AC's. In step 4 above we replace the \$AC's (encr. \$32's) with \$AE's (encr. \$30). It works!

ADDENDUM:

The procedure given last month for making an archival backup of SKY TRAVEL also works on another Commodore program, BGRAPH. Just make the same changes indicated for SKY TRAVEL, except make them to track 5, sector 7 on a copy of BGRAPH. The byte locations and values to use are the same on both programs. This procedure may also work on some of Commodore's other programs. Look for a program which knocks the head three times when it loads. Another thing to look for is a file name that looks like a long underline. The sector to change on SKY TRAVEL was the first sector of that file.

The following was submitted by a Newsletter subscriber:

The program is **SNOKIE**, by Yves Lempereur and Troy Lyndon, (C) 1984 Funsoft, Inc.

TYPE OF PROTECTION: Track 2 contains a read error 21 (no sync). The program checks for the presence of specific bytes on track 35, block 0 and uses an autoboot and machine language encryption. There are other minor check routines including changes to track and sector pointers to inhibit file copying.

HOW TO COPY: Use a copy program that will place the error 21 on the clone disk for you.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

1. Copy with any copy program which doesn't put errors on the disk.
2. Load and run a machine language monitor at \$C000. Start it with SYS 49152.
3. Load the boot with L"SNOKIE",08. This is an 8 block autoloader occupying computer memory from \$0326. After a few seconds, you will hear the read/write

head move to track 35 - at this point press RUN/STOP. Enter the monitor with SYS 49152 again; you now have the entire loader in memory.

4. Using the M command, display 8 bytes from \$0858. Cursor to the first byte (\$D0) and type over it with \$F0.
5. Again using the M command, display 16 bytes (2 lines) from \$0398. Make the following changes:

FROM: \$A9 00 8D 98 08 A9 15 8D 99 08

TO: \$EA EA EA EA EA EA EA EA EA EA

6. Using M \$0884, change the bytes \$05, \$4B, \$23, \$6E to some other value (e.g. \$99, \$99, \$99, \$99).
7. S"@0:SNOKIE1",08,0326,0A9B
8. That's it - you're done.

Prior to making the above changes, it is a good idea to print out a disassembly of this program to examine the many varied M/L routines that are used. You will find 2 block-reads (U1), a block-write (U2), numerous examples of the techniques involved in setting up files and commands for loading/saving information etc.

The byte changed in step 4 above replaces a Branch if Not Equal instruction (BNE) with a Branch if Equal (BEQ). This is contained in a routine which compares 4 specific bytes, read via a U1 command from track 35/0, to \$05, \$4B, \$23, and \$6E. If any of these bytes were read as different, the program branches to an endless loop at \$0862.

The 10 bytes changed in step 5 above remove (NOP) instructions to store specific values at \$0898 and \$0899. These values are placed strategically in C64 memory to show up as track and sector pointers. Being "spurious", their presence inhibits the use of a file copier to copy the main program "SNOKIE". Examination of a disassembly printout will reveal other routines from which a great deal may be learned.

The bytes changed in step 6 above replace the expected values returned by the U1 from sector 35/0, with an uncommon value which is unlikely to be met, i.e. \$99. You may, if you wish, use a track and sector editor to change those bytes in 35/0 to \$01.

There is an alternate method of making an archival backup of this program. However, it is suggested that this be done at a later time so as to benefit from examination of the original autoloader.

1. Load and run a \$C000 monitor as before.

2. L"SNOKIE",08 and let the program execute.
3. Press your RESET button and type SYS>49152.
4. With the main program (SNOKIE1) still in memory, save it to a separate formatted disk: S"SNOKIE",08,1000,A000.
5. That's it - you're done. The program may now be loaded with LOAD "SNOKIE",8,1 and executed with a SYS 28990.

The original autoloader program provided an encryption value used by the main program. In the future, you will now LOAD a decrypted copy. The SYS command jumps to the entry point at \$713E (28990), loads a HIGH SCORES module (copied separately from the original disk), and executes correctly. A simple BASIC loader may be created to load and run the main program for you:

```
10 IF A THEN SYS 28990
20 A=1 :LOAD "SNOKIE",8,1
```

The program is **SPACE TAXI**, Copyright 1984, John F. Kutcher. Published by Muse Software.

TYPE OF PROTECTION: The disk contains a error 29 on track 3, sector 1.

HOW TO COPY: Use any copy program that can copy error 29's.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

1. Make a copy of the original disk with no errors. You can file copy the disk if you wish.
2. Load and execute a monitor at \$8000 or above, such as HIMON.
3. Insert the copy disk and load in the TAXI/CMD file with L "TAXI/CMD",08
4. Change location \$5F0D from \$F0 to \$D0. This will change a BEQ to a BNE.
5. Save the file back to the copy disk with S "@0:TAXI/CMD",08,4000,7530.
6. That's it - you're done.

The protection code in this program can be found relatively easily by tracing through the boot process. The process is started with LOAD "*",8,1. This loads in the file "SPACE TAXI" at \$00C6-0829. Since the file starts so far down in memory, there are several possible ways this program might autostart itself (see the "AUTOBOOTS" chapter in PPM Vol. II). Luckily, the \$00C6 starting address is a dead giveaway. This is the location which specifies the number of characters in the keyboard buffer. Why should the file start loading here? To find the

answer we need to be look at the contents of the file. The standard technique with an autostart routine is to use a track/sector editor to change the load address of the file directly on disk. The "SPACE TAXI" file starts at sector 17/10. Bytes \$02-03 of this sector specify the load address in lo-byte/hi-byte order, \$C6 00 (\$00C6). If we change the \$00 to \$10, the file will load at \$10C6, preventing it from autostarting and allowing us to examine it.

If we alter the load address, load the file and examine location \$10C6, we see that it contains a \$06 (remember, this will actually be loaded into \$00C6). This indicates that there are 6 characters waiting in the keyboard buffer. Since the keyboard buffer is normally located at \$0277-80, we need to examine locations \$1277-80 in our altered version. Sure enough, looking in this area reveals 6 characters there, a "RUN" command plus a couple of cursor returns. The next time keyboard data is requested, these characters will be provided. This will happen when the original LOAD:*,8,1 command is completed executing. The BASIC interpreter will automatically start looking for characters typed from the keyboard, and so it will see the "RUN" command. Where there's a "RUN" command there must be a BASIC program. If we look at the part of the file that will be loaded into the BASIC area (normally at \$0800, but at \$1800 instead in our altered version), we see a short BASIC program: 10 LOAD "TAXI/LOADER",8. This is the next step in our boot process. Load TAXI/LOADER and look at it. All this BASIC program does is put up a picture of the space taxi, LOAD "TAXI/CMD",8,1 and then SYS 16384 (\$4000). From the monitor, load "TAXI/CMD". Location \$4000 has a JMP \$5E9F statement. Disassemble starting at \$5E9F. After a subroutine call to initialize the VIC chip, some other initialization is done, including one important step: an \$A9 (LDA) is stored into location \$5D3D.

Next, a subroutine at \$6F55 is called. This begins a series of program loads, using the normal KERNAL routines to load in the files "TAXI/VOICE", "TAXI/VOC", etc. Still in the \$6F55 subroutine, another subroutine at \$5D3D is called. This reveals the main protection routines at last. Recall that an \$A9 was just stored into \$5D3D. This disassembles as LDA#\$60, followed by STA#\$5D3D. The program is 'covering its tracks' by putting an RTS (\$60) instruction at \$5D3D. Next, the error/command channel (15) is opened, then a random access channel is opened as file 5 (like OPEN 5,8,5,"#" in BASIC). This will allow the program to send BLOCK-READ (B-R) commands to the disk to check 3 sectors: sectors 2/1, 3/1 and 4/1. Sector 3/1 contains an error 29 but the other two do not contain any errors. Each sector is checked in the same way. An address pointer is stored in \$00FB-FC in lo-byte/hi-byte order. Then a subroutine at \$5DD2 is called. It starts by setting the command channel for output and clearing the Y-index register to zero. Using location \$FB-FC as a pointer indexed by Y, it then sends the bytes of the B-R command. The B-R commands have been encrypted, however, so each byte is decrypted by subtracting \$02 from it before being sent to the drive. The end of the B-R command is reached when a \$00 byte is encountered. This ends the subroutine at \$5DD2.

Next, a subroutine at \$5DF0 is called. After setting the error channel for input, it reads in the error message from the last B-R attempt. The entire message is stored starting at \$033C and then the error channel is closed. Now comes the tricky part. The two bytes of the error number, at \$033C-3D, are

combined mathematically into a single byte. Depending on which error number, 29 or 00 (no error) is returned, a different value will be produced. This is a good excuse to learn about the ADC, ASL and SBC statements used in this process. ADC (ADD with Carry) takes the accumulator (A) and adds to it the value specified AND the carry bit. If the sum is over \$FF, the carry is set to 1; otherwise it is cleared to zero. SBC (SUBtract with Carry) takes A and subtracts from it the value specified AND the borrow bit. The borrow bit is simply the opposite of the carry bit: when the carry is 0, the borrow is 1 and vice versa. If the result of the subtraction is less than zero, the borrow bit is set; otherwise the borrow is cleared. ASL (Arithmetic Shift Left) shifts the bits of A left one position. The original leftmost bit is shifted into the carry bit, and a zero is shifted into the rightmost bit of A. This has the net effect of multiplying A by 2.

At \$5E19, we load the second byte of the error number into A. This is either "0" (00,OK) or "9" (error 29) in ASCII. In hex, it is either \$30 or \$39. After clearing the borrow by setting the carry with SEC, \$30 is subtracted from A to give \$00 or \$09. Two ASL's in a row multiply A by 4 to give \$00 or \$24 (36). This is stored in \$FB. Another ASL gives us \$00 or \$48 (72). After clearing the carry with CLC, location \$FB is added in, yielding \$00+\$00= 00 or \$48+\$24= 6C (108). The carry is cleared again and the first byte of the error number is added in, either \$30 ("0") or \$32 ("2"). The result is \$30 or \$6C+\$32= 9E (158). Finally, the borrow is cleared (SEC) and the value \$2E (46) is subtracted from A to give either \$30-\$2E= 02 or \$9E-\$2E= 70 (112). This final result is left in A when the subroutine at \$5DF0 ends and returns to the \$5D3D subroutine. Remember, a \$02 is returned when there is no error, while a \$70 is returned in the case of error 29. The error numbers from all three sectors are encrypted this way. The first two results (\$02 and 70) are stored in \$5DD0-01, while the third is left in A (\$02). The three bytes are then encrypted together to produce one 'magic' byte. This is done by the code at \$5D98. The third byte is doubled with ASL to give \$04, then the carry is cleared and the second byte (\$70) is added to give \$74. This is doubled again to give \$E8, then the first byte (\$02) is added to give \$EA. This is the 'magic' byte, and it is stored at \$11.

This is the end of the \$5D3D routine, which returns back to the \$6F55 routine. This stores a few unrelated values and then returns to the \$5E9F routine (main initialization). A series of unrelated subroutine calls follows, which would take a lot of time to trace before verifying that they are not related to the protection scheme. By looking ahead a bit however, we'll see a piece of code at \$5F07 which checks location \$11 for the proper value. It does this adding the value \$16 to the 'magic' value \$EA. The sum is \$00 when the proper error was found. A BNE (Branch if Not Equal to zero) statement is used to branch if the sum is not correct. This prevents the normal first screen from being loaded if the protection does not pass. A demo screen is loaded instead. In step 4 of the archival procedure, we change the BNE statement to a BEQ (branch if EQUAL) so that a copy disk will pass.

A few more words are in order. We could have skipped going through the calculations if we had a way to save the contents of \$11 after we loaded the original disk and hit RESET. This is easy with a special-purpose cartridge, but difficult otherwise since the RESET routine clears most of memory from

\$0000-0800. This gave us the incentive to learn about ADC, ASL and SBC. We could have bypassed the protection by inserting code to store the proper value in \$11. By looking around a bit, we have found a more 'elegant' way to bypass the protection by changing only a single byte.

The program is **SPELLICOPTER**, Copyright 1983, DesignWare, Inc.

TYPE OF PROTECTION: The program will check TRACK 1 SECTOR 1 for an error 23.

HOW TO COPY: Copy the program with any copy program that can place an error 23.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

Since this program is stored in user files, we will use the back door to unprotect it.

1. Make a copy of the original disk without errors. Format another disk to store the program files we will create.
2. LOAD and RUN the program from the original disk.
3. When the menu screen appears, RESET your computer.
4. LOAD and execute HIMON with SYS 49152.
5. We will not save the new program file to the original disk, because we will run the risk of writing over data that is needed for the other functions of the program (REMEMBER NEVER MODIFY YOUR ORIGINAL DISK). We will save the first section of code to the formatted disk through HIMON.

Flip out BASIC with your M Command at 0001. Change the 37 to a 36. Save the first section of code with S "COPTER1",08,0800,BFFF.

6. Since HIMON wrote over our program code at C000-CFFF, we will have to LOAD the original disk again. Power down and LOAD and execute the original program.
7. When the menu screen appears, RESET your computer. LOAD and execute LOMON with SYS32768.
8. Save the code at C PAGE with S"COPTER2",08,C000,CFFF. Remember, save the code to your formatted disk not the clone disk.
9. Now for the entry point. Using the D command at \$0800, we find a JMP to \$08DB. This section of code loads in the main menu screen.
10. Power down and load in the two sections of code.

LOAD"COPTER1",8,1 (RETURN)

NEW (RETURN)
LOAD"COPTER2",8,1 (RETURN)

11. With the two sections of code in memory, insert your copy disk and type
SYS 2267 RETURN. This is the decimal equivalent for \$08DB.
12. YOU'RE DONE!

The only real trick to this program was finding the entry point. A JMP instruction is always a good place to try when looking for an entry point. If a JMP does not work there are other things to look for.

You will find a more complete explanation of entry points in the PROGRAM PROTECTION MANUAL VOLUME II.

When looking for an entry point, be adventurous. You can't harm your computer. You may have to power down and begin again, but all you lose is time. Keep notes on what you have tried. Good notes can save you a lot of time.

You may find that an entry point appears to work, but you get garbage on the screen. This may be the result of your monitor overwriting a section of code needed for the proper execution of the program. Before you begin your hunt for a new entry point, try loading the program you are working on in the normal manner. Stop the program and try your entry point without loading a monitor. If this works, you need only save out the code in sections as we did with our program.

The program is **SPELLMASTER**, COPYRIGHT 1982 BY SYSTEMS SOFTWARE

TYPE OF PROTECTION: This program will check TRACK 1 - SECTOR 7 for an error 22.

HOW TO MAKE COPY: Copy the program with any good copy program and place an error 22 on track 1 sector 7, or use a copy program that will place the error for you.

HOW TO MAKE A WORKING COPY OF THE PROGRAM WITHOUT ERRORS ON THE DISK:

- 1). First make a copy of the disk without errors.
- 2). The program that will do the error checking is called "SPLA105203PP". This program is located in memory from \$0800 to \$2A00.
- 3). Load and execute LOMON with SYS32768. Provide a clean work space, with the fill command (F 0800 3000 00).
- 4). L "SPLA105203PP",08. Using the I command, we locate the UI command at \$0F33. As expected, it points to track 1 - sector 7. This program will branch a great deal, but it all comes down to removing a subroutine.
- 5). Using the M command, change the 20 (JSR) at \$16AA to a 60 (RTS).

6). With this change, S "@0:SPLA105203PP",08,0800,2A00.

7). YOU'RE DONE.

As you probably noticed, we did not tell you much about this program. We include this as an example of our "give it a try" philosophy. We traced this program through attempting to discover exactly how the programmer was protecting it, but to no avail. The programmer had so many JSR's in this program that we found it extremely difficult to trace. We did learn that \$16AA seemed to be an active subroutine. Since files were being manipulated in this section, we thought this a likely place to start.

Don't be afraid to try something. Remember the only thing you will lose is a little time. If it doesn't work, load the program and try again!

The program is SPY VS SPY, Copyright 1984, 1st Star Software.

TYPE OF PROTECTION: This disk is littered with errors. The program will rattle your drive twice before the program begins.

HOW TO COPY: Use OMNICLEONE to copy the disk or any other program that can handle errors of this type.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

- 1). This program places the directory in an endless loop. We will fix this problem first through a track and sector editor. Track 18 Sector 01 contains the pointers for the next block of the directory in the first two bytes of the sector. At 18/01, we find the pointers directing us to 12/01. 12/01 is the HEX value for 18/01. Since this is the only block of the directory, this should read 00FF telling the disk drive that this is the last block of the directory. In HEX mode, change the 1201 to 00FF.
- 2). Now for the protection. Load LOMON and execute with SYS32768. The program that contains the protection is called "S0". L "S0",08. This program is located in memory from \$6000-6250. Using the I command, you will locate two U1 commands from \$6180-\$61C0. When we switch to the D command, we locate CMP #32 at \$61F2 followed by a BNE \$61FE. If you have studied the PROGRAM PROTECTION MANUAL VOLUME I or the NEWSLETTER, you know how to deal with CMP#\$32's. You either change the compare or the branch instruction. This is easy enough, but when you look further in the code you find another compare instruction. \$61F9 has a CMP \$FF followed by the same branch instruction (BNE \$61FE). The CMP \$FF instruction will compare the value that was returned from the error channel and stored at \$00FF. A reset will erase this value. In this program, we must change the branch instruction to defeat the protection scheme.

- 3). Using the M command at \$61F4, change the D0 to an F0. This will take care of the CMP#\$32. Now use the M command at \$61FB to change the next branch from a BNE to a BEQ by changing the D0 to an F0.
- 4). Save out the altered code with S "@0:S0",08,6000,6250. That will do it.
- 5). YOU'RE DONE

The program is SUICIDE STRIKE, Copyright Tronix., 1984.

TYPE OF PROTECTION: This program checks track 1 sector 1 for an error. If the error is present, the program will run properly.

HOW TO COPY: Copy the disk with any good copy program and place an error 23 on 01/01. You may also use a nibble copy program and let it put the error on for you.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK.

1). PROBLEM

The program uses illegal characters in the file names. Since basic will not recognize these characters, we will be unable to load the programs into a machine language monitor to alter the error checking routine.

SOLUTION

Load your track and sector editor, so that we may alter track 18/01. In ASCII mode it would appear that all three program files have the same name (""). If we change to HEX mode, we will notice a difference.

Let's review a bit from P.27 in the P.P.M. 18/01 contains the list of the file names on a disk. Looking at 18/01 we have the following information.

00-FF	- last block of the file
82	- program file
11-00	- this file is located on track 17 sector 0.
22-03-05-0F	- this is the name of the first file.

Let's alter the name of this program. Since it's only purpose is to load in the second program, it doesn't matter what we call this program. Let's switch to ASCII mode, and change the name to SUICIDE BOOT. Begin at byte 05 and type in the new name.

2). PROBLEM

The second program name also utilizes illegal characters. This program contains the error checking routine. If we change the name at 18/01, we must also

change the name in the first boot program, or when it tries to load this program we will get a file not found error.

SOLUTION

The second program is called 22-03 in HEX. Let's change that to 53-32 in HEX at 18/01. Now we must change the name in the first boot program. Go to 17/00 and locate the original program name (22-03) and change it to 53-32 (S2). Now when we run SUICIDE BOOT, it will be looking for S2.

3). PROBLEM

When the second program is run, it will be looking for a program called 22-09 in HEX.

SOLUTION

Let's use the same procedure as Step 2 to correct this problem. Go to 18/01 and change the third file name to 53-33 HEX (S3). Now locate the first block of the second program, which is located at 19/00. Find the 22-09 and change it to 53-33 (S3).

Now that the names have been changed, we will be able to load the S2 file through a monitor and alter the error checking routine. We'll get to that in a moment. While we're still at 19/0, let's take a look at the code in ASCII. We see a B-R at 01/00. This is the error checking routine that we are looking for. When we go to HEX, we see the kernal calls. Remember they are stored low byte - high byte (CF FF). Further investigation, reveals the CMP 30. This is stored on the disk as C9 30. C9 is the machine language instruction for compare. If we change the 30 to a 32, the program will crash if it finds an error. This will fix our problem.

There may be times when it is easier to alter the code on the disk. The machine language instructions are the same as you will find in a disassembly of the code. You just have to get used to the way the code is stored. When we look at a disassembly of a program, we find the assembly code on the right and the machine code on the left. Get used to working with both.

EXAMPLE

MEMORY ADDRESS	MACHINE CODE	ASSEMBLY CODE
08D3	20 CF FF	JSR \$FFCF
08D6	C9 30	CMP #\$30

The middle column is what we would find on the disk, which is similar to what we find using the M command in a machine language monitor. We are missing the memory address and assembly code. Using a list of OPCODES, we may still follow the logic of the program and alter the code as necessary.

ALTERNATE SOLUTION

If you feel more comfortable altering the code through a monitor, make the changes that follow:

- 1). Change the program names at 18/01 to SUICIDE BOOT, S2, and S3 respectively.
- 2). Go to 17/00 and locate 22-03. Change this to 53-32 (S2).
- 3). Go to 19/00 and locate 22-09. Change this to 53-33 (S3).
- 4). Load and execute HIMON. Fill the memory from 0800 to 9FFF with 00
- 5). L "S2",08 - This is the program that contains the error checking.
- 6). Using the I command, locate the B-R.
- 7). Using the D command, disassemble the code at 08D0. At 08D6, you will find the CMP 30. The program is checking to see if 01/01 is a good block. If it is, the program will branch and crash. If we change the CMP 30 to a 32, the program will branch if it finds a bad block. Using the M command, change the code at 08D7 to a 32.
- 8). S "@0:S2",08,0800,0A00
- 9). YOU'RE DONE

The program is TRIO, Copyright 1984 Softsync Inc.

TYPE OF PROTECTION: Error 23 on block 5/19 and error 20 on block 5/20.

HOW TO COPY: Any nibble copier will copy this disk.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

1. Make a copy of the disk without errors. Through BASIC, LOAD "TRIO FILE",8. List lines 1-200. There you have the error routine. Line 30 sets up the check of block 5/19. In line 40, there is a GOSUB 900 for the INPUT routine. From there we check for the error 23. If not found, we will branch to 1000. The 1000 subroutine will give us another chance to insert the disk containing the errors. If, after the second try, the error is not found, we will be sent to a warm reset (SYS 64738). Line 50 sets up the check of block 5/20 for an error 20 and follows the same procedure as the previous check.
2. To save time in the program load, we will remove the entire error checking routine with the following changes:

DELETE lines 10,20,30,40,50, and 70

3. Save out the altered code to the COPY disk with SAVE"@0:TRIO FILE",8.

4. Follow the same procedure for the files "TRIO WORD" and "TRIO CALC".
5. That's it - you're done.

This program uses an autoboot at \$02A7 to load and run a BASIC program. Autoboots of this type are explained in PPM VOL II, but this one has an interesting twist so we'll trace the code.

1. Load and execute HIMON. Load the autoboot from HIMON with L "TRIO",08.
2. Using the D command, disassemble the code at \$02C3. There we find the location of the file name, which is at \$02A7. Using I \$02A7 will reveal the program name. If you print out the directory, you'll find this program in your listing beginning with an uppercase R followed by RVON, RVOFF, D, RVON, RVOFF, M. This is the BASIC program that is used to load the other three modules.
3. Once it gets the program, a RUN command is inserted into the program code through the routine at \$02DF. At \$02DF the program places these characters into the input buffer (\$0277-80). From here the program jumps to BASIC's input routine (JMP \$A483). When BASIC looks for input, it will see the RUN command and execute it.

What we have here is an attempt to confuse us. You can get a look at this hidden program with LOAD "R*",8. It will list and run normally. All this effort to hide a program that does not even contain the error protection. The protection code is contained in the programs called TRIO CALC, TRIO WORD, and TRIO FILE.

The program is TRIVIA, COPYRIGHT 1983 CYMBAL SOFTWARE INC.

TYPE OF PROTECTION: The main program is stored on the disk in user files. The loader routine will load the user files into memory and store the files at the proper memory location in the computer. The loader program will also check for an error on the disk. If the proper error is found the program will execute.

HOW TO COPY: Use any good copy program and place the errors on the proper track and sector. You may also use one of the newer nibble copy programs to copy the disk and place the errors on the disk for you.

HOW TO MAKE A WORKING COPY WITHOUT ANY ERRORS ON THE DISK.

The techniques used on this program are described in pages 45-50 of the Program Protection Manual. You will be able to apply the information contained here on many different programs.

- 1) Copy the original disk with BACKUP 228 or any good copy program that will not

place any errors on the destination disk.

- 2) Load and run the original disk. As the program executes you will find that the prompts react extremely slow. When you press a key it seems to take forever for the program to respond. This is a very good indication of a program that is written in BASIC: slow response time. Compare the response time needed for this program to your favorite arcade type game. Extremely slow programs will be written in BASIC.
- 3) With the program in memory, RESET your computer (page 45 of P.P.M.). Load and execute the program called RESTORE from your disk. Now list the program. You have just captured the whole program in memory.
- 4) Save the program out to a newly formatted disk (not the copy disk). The copy disk was only an insurance policy, always make a copy of every disk prior to doing anything.
- 5) Repeat the same procedure on the other modules of the program. You may now save all four parts of the program out on one disk.
- 6) YOU'RE DONE.

The program is **EXODUS: ULTIMA III**, Copyright 1983, ORIGIN SYSTEMS.

TYPE OF PROTECTION: The program is looking for an error on track 6. If the error is present, the program will run properly. This program is stored on the disk in user files, making it difficult to examine the code. If you have one of the new track and sector editors that include a disassembler feature, the task is made easier. It is possible to do the job without a program of this type, but it is a bit more time consuming.

HOW TO COPY: Make a copy of the original disk with any good copy program and place an error 23 on track 6.

HOW TO MAKE A WORKING COPY OF THE PROGRAM WITHOUT ERRORS ON THE DISK:

- 1). Make a copy of the original disk without errors.
- 2). When we run this program from the original disk, we find that it reads the error channel twice. Load the program normally from the original disk. After approximately 15 seconds, reset your computer. Load and activate LOMON with SYS32768. If we disassemble the code at 09BC, we find the familiar kernal calls. At 09C4 and 09CB, we find the two comparisons for 30. The first comparison checks for a good block. If a good block is found, we will branch to a dead end. If a bad block is found, we JSR \$FFCF. This is the command to input a character from a channel. This is the second read of the error channel. If a good block is found, we are sent to the section of code that will crash the program. If this were a program file, we would change the 30's

to 32's, so that the program would crash on a bad block. We would then save the corrected program file. Since this program is stored in user files, we will have to make these changes on the disk.

- 3). With your track and sector editor running, let's examine track 12 sector 0. In ASCII mode, you will find a U1 command. If we begin interpreting the code on the disk at 12/00, we find the CMP#\$30's.
- 4). Let's get around the protection by changing the CMP#\$30 to CMP#\$32. The program will now be checking for bad blocks. If one is found, the program will not run properly.
- 5). In HEX mode, find the two spots that compare for 30. The code you are looking for is C930 (CMP#\$30). Change the 30 to a 32 in BOTH places on the block.
- 6). Use your write function to write these changes to the disk.
- 7). YOU'RE DONE.

The following was submitted by Tony & Bubba, Schenectady, NY.

The program is **ULTRACOPY-II** by Jim Lagerkvist. Copyright 1985.

TYPE OF PROTECTION: Track 1 is heavily protected. Most copy programs will hang up on this track.

HOW TO COPY: Some of the new parameter copiers may be able to make a working copy.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS:

1. Make a copy of the disk with a copy program that will let you copy just tracks 2-35 (don't copy track 1). You'll probably need to use a nibble copier to do this.
2. Recopy the first copy with a 3-minute type copier that will not put errors on the disk. You can now disregard your first copy, it is no longer needed.
3. Load the program from the copy disk with LOAD>"ULTRACOPY-II",8,1 and let the program crash. It will take you back to a warm start or say it can't run on this configuration. At this time you must RESET your computer. You must RESET even if the program takes you back to a warm start.
4. Load in HIMON or other monitor at \$C000. Enter the monitor with SYS 49152. Using the command M 2BD7, look for a \$94. Change the \$94 to a \$90. This program was compiled with Petspeed. The \$94 is a branch if equal. The \$90 is a branch if not equal to.

5. On RESET you lost the first few bytes of the BASIC program, so we have to make sure the right bytes are at \$0800. Using the command M 0800, you will see \$00 00 00. Change this to \$00 0B 08.
6. At this point the copy disk is full and we can't save the program code back onto it. Save it to a formatted disk (not the copy disk): S"TEST",08,0801,3501
7. Load in a track/sector editor and look at track 18 sector 1 of the 3-minute copy. The first three bytes will be \$12 01 F8. The \$12 01 points the directory to track 18 (\$12) sector 1. This keeps the directory in an endless loop. Change these three bytes to \$00 FF 00. The \$00 FF now says this is the last sector of the directory. The \$82 unlocks the program ULTRACOPY-II so we can scratch it and save our new code back on top of it. While you have the T/S booted up, you might want to try fixing up some of the other directory entries (see PPM Vol. I).
8. Exit the T/S editor and scratch the program from the 3-minute copy disk:
OPEN 15,8,15, "S0:ULTRACOPY-II" :CLOSE15
9. Reload HIMON and load in the program we saved before: L>"TEST",08. When ready, save the code back to the copy disk as follows:
S "ULTRACOPY-II",08,0801,3501
10. That's it - you're done. LOAD"ULTRACOPY-II",8 and RUN to execute. The program uses other info on the disk so you can't file copy this disk.

The program is **UNGUARD**, Copyright 1983 by MICRO-WARE

TYPE OF PROTECTION: The disk uses multiple errors on one track. When the track is checked with the 'U1' command a specific type of error is returned. The program also uses an ML DOS (Disk Operating System) routine to check for the proper errors. When the ML DOS routine finds the proper error(s) (as on the original) it will store a value in an unused memory location of the disk drive. If the error is not present the ML DOS routine will place another value in this memory location of the drive. When the main program executes, it will use the value stored in the disk drive as a check to see if the disk was the original or not.

The way to tell if a program was using any values stored in the disk drive is to use two disk drives. Hook up one drive at this time.

- 1) Load and execute the program. Use the program to verify the proper operation of the program.
- 2) Once you have verified that all the functions of the program operate properly then turn the power on to the second drive. Carefully disconnect the serial bus from the first drive and connect it to second drive. Do not turn off the power to the first drive.

- 3) Use the program and verify that the program does NOT operate as expected. If any values were stored in the disk drive the program will not operate properly.
- 4) Carefully hook the serial bus back up to the first drive and use the program. This will confirm that the program actually stored some data in the disk drive that was necessary for the proper operation.

HOW TO COPY: Some of the newer (and better) 'nibble' copy programs will successfully copy this program.

HOW TO MAKE A WORKING COPY WITHOUT ANY ERRORS ON THE DISK.

The main part of this program is written in BASIC. There is also ML code stored from \$C000 to \$D000. This ML code is the error generating routine(s) that will be stored in the disk drive to actually do the error generating. In examining this code it becomes apparent that each error generating routine will do an EXCLUSIVE OR (EOR) with memory location \$14 (decimal 20). This is the location where the proper value was stored in disk drive's memory. By simply loading and executing the program one may obtain the proper value that was placed in the disk drive's memory. After the program is up and running disconnect the serial bus and turn off the computer. Then do a memory read (M-R) of the desired location (20 decimal). The value returned (220 decimal) is the key to making this program work.

- 1) Load and execute the main program. After it is up and running RESET your computer.
- 2) LOAD "RESTORE",8,1 and press RETURN. Then SYS525:CLR (use the program UNNEW if you wish)
- 3) You may now list the BASIC portion of the program. Delete line 132. This is the routine that loads the ML code at \$C000 and checks for the proper errors.
- 4) Add the following lines of code to the beginning of the program:

```
10 IF A THEN 30
20 A=1:LOAD"C0 D0",8,1
30 OPEN15,8,15
40 PRINT#15,"M-W";CHR$(20)CHR$(00)CHR$(1)CHR$(220)
50 CLOSE15
```

Line 10 is a trap to load in the code from \$C000 to \$D000.
 Line 20 sets the trap and loads the code.
 Line 30 opens the error/command channel to the drive.
 Line 40 stores the proper value in the drive at the proper location.
 Line 50 closes the error/command channel.

- 5) Save the modified program out to a newly formatted disk.

- 6) Load and execute the LOMON. (ML monitor residing at 32768).
- 7) Save the code from \$C000 to \$D000 on the same disk as you saved the modified program to.

S "C0 D0",08,C000,D000

- 8) YOU'RE DONE ——— More and more it is becoming necessary to look in the drive for data.

The program is UP'N DOWN (TM), Copyright Sega, 1984.

TYPE OF PROTECTION: The disk contains an error 23 on tracks 31-35. The program will check one of these tracks for this error and run if present.

HOW TO COPY: Make a copy of the disk and place these errors, or make a copy with a nibble copy program that will place the errors for you.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS:

- 1). Make a copy of the program without errors on the disk.
- 2). The program called BOOT contains the error checking routine. Since this is an auto boot, it will be difficult to examine the code without loading it to another area of memory. Load and execute LOMON with SYS 32768. Reset your computer with G FCE2. Now load the 1st boot into BASIC with LOAD "AUTOLOAD",8. Re-activate LOMON with SYS 32768. This will defeat the auto load and allow us to examine the code. Disassemble the code a \$0801. It would appear from the program code that there is nothing unusual about the code. Using the D command at \$0868, you will find a JMP \$4700. This is the starting address for another program on the disk called "LOADER". Further on, we will find a JMP \$414F, which is a jump to "never-never-land". This leads us to believe that once the program has found its error, it will continue with normal program execution, by loading and executing the "LOADER" program. Let's give it a try.
- 3). LOAD "LOADER",8,1 from the copy disk. Now all we need do is convert this program's starting address to decimal. This will give us an address of 18176. With the program in memory, type SYS18176 followed by a return.
- 4). YOU'RE DONE.

Remember no matter how cryptic the code may seem, the result is always the same. Don't be afraid to experiment. You can't hurt your computer, but you will save your disk drive.

The program is VIP **TERMINAL** (TM), Copyright 1983 by Softlaw Corp.

TYPE OF PROTECTION: The disk contains an ERROR 29 on TRACK 1 and 2, and an ERROR 23 on TRACK 18,6. The format code "A" on TRACK 18,0 / BYTE 02 has also been altered to prevent writing to the disk (see PPM Vol I for a discussion of the format code). The program will check for the presence of all of these errors.

HOW TO COPY: Use any copy program capable of placing these errors.

HOW TO COPY WITHOUT ERRORS ON THE DISK:

This program is stored on the disk in user files, which are loaded by a machine language loader. The loader is located on the disk on TRACK 19, SECTORS 0-6. We will make our modifications directly on the disk.

1. Make a copy of the disk without errors. Next, the format code on TRACK 18/0 must be corrected so you can write on the disk. You need to change BYTE 02 from an \$01 to a \$41 (DECIMAL 65, "A"). Since the disk cannot be written to by ordinary means yet, you must use the UN-WRITE PROTECTOR program given below to make this change.
2. With the format code changed, we will now make the other changes necessary for the proper execution of this program.
3. Using a track and sector editor, make the following changes:

TRK/SEC	BYTE LOC.	ORIGINAL	MODIFIED
19/0	7 & 19	32 (both)	30 (both)
19/1	123-125	20 27 1A	4C EC 18
19/1	196 & 208	32 33	30 (both)

NOTE: Byte locations are given in decimal; bytes are counted starting from 0, not 1.

4. The loader is designed to select and load a series of programs. Since the disk only contains the VIP **TERMINAL**, press return when the "DESKTOP" screen is displayed. The program will load in about 2 minutes.
5. YOU'RE DONE!

UN-WRITE PROTECTOR

```
10 OPEN>15,8,15,"I0"
20 OPEN>2,8,2,"#"
30 PRINT#15, "U1:2 0 18 0"
40 PRINT#15, "B-P:2 2"
50 PRINT#2, "A";
60 PRINT#15, "M-W" CHR$(1) CHR$(1) CHR$(1) "A"
70 PRINT#15, "U2:2 0 18 0"
80 CLOSE 2
```

90 CLOSE 15

UPDATE on VIP TERM (tm) , Copyright 1983 by Softlaw Corp.

1. Start by making a 3-minute copy with no errors. Run the UN-WRITE PROTECT program below on the copy (be sure to use a ";" at the end of line 20).

```
10 OPEN 15,8,15,"IO" :OPEN 2,8,2,"#"
20 PRINT#15, "U1:2 0 18 0" :PRINT#15, "B-P:2 2" :PRINT#2, "A";
30 PRINT#15, "M-W" CHR$(1) CHR$(1) CHR$(1) "A"
40 PRINT#15, "U2:2 0 18 0" :CLOSE 2 :CLOSE 15
```

2. We found two other versions. Load a T/S editor and examine your disk for ONE of the following sets of information. Then make the changes indicated for your version.

a) Sector 19/00: change byte number \$08 from \$32 to \$30; change byte \$14 from \$39 to \$30.

Sector 19/01: change byte \$77 from \$04 to \$2B; change byte \$BF from \$32 to \$31; change byte \$CA from \$37 to \$30.

OR b) Sector 23/01: change bytes 00-06 from \$CF 3E 98 3F 35 9F B7 to \$7D 3E C3 1A 50 EE F7; change byte CF from 97 to D6.

The program is THE VISIBLE SOLAR SYSTEM, Copyright 1982, Commodore.

This program may be backed-up through the CARTRIDGE BACKER program available through CSM SOFTWARE INC. Those who own the CARTRIDGE BACKER, may skip the backup procedure, but we do recommend that you read through the explanation. We chose this program for study, because the backup process illustrates some interesting concepts. This technique will require a CARTRIDGE BACKER BOARD, the CARDCO 5 EXPANSION INTERFACE, or any cartridge board that allows control of the GAMEROM and EXROM lines.

The GAMEROM and EXROM lines allow us to chose different memory configurations for the computer. On power-up or reset, the computer will check these lines to determine which memory map is being requested. In order to gain access to the code for this cartridge without activating the program, we must be able to manipulate these lines.

An ULTIMAX cartridge grounds the GAME line, which flips out the KERNAL and the cartridge area from \$8000-A000. These cartridges are usually 8K and operate in the KERNAL area (\$E000-\$FFFF). As with the BASIC, there is a section of RAM beneath the KERNAL ROM. Storing a 36 at \$01 will flip out BASIC. To utilize KERNAL RAM, a 35 must be stored at \$01.

Let's get to the backup process.

- 1). Load and execute LIMON with SYS 8192.

- 2). If you are using the CARTRIDGE BACKER BOARD, insert the board into the computer with the switches and computer off. Turn on the computer and turn on the switches in the manner described below. Insert the cartridge into the board. If you are using the CARDCO BOARD, insert the cartridge in one of the slots and set the switches as indicated below.

CARTRIDGE BACKER BOARD

- 4 - POWER SWITCH
- 2 - EXROM
- 1 - GAME ROM

CARDCO BOARD

- MASTER CONTROL RIGHT SWITCH ON
- MASTER CONTROL LEFT SWITCH ON
- RIGHT SWITCH FOR CARTRIDGE SLOT BEING USED ON

You must use the order given above or the computer will lock-up. If this should occur, power-down and start over.

- 3). With the switches activated, we will now flip-out the BASIC ROM. Using the M command at 01, change the 37 to a 36.
- 4). You will find the code for the cartridge stored from \$A000-BFFF, but we are not done yet. The cartridge cannot be run from this area of memory, so we must now add a section of code to transfer our code from \$A000 to \$E000 and \$2000. An ULTIMAX cartridge requires an image of the program to appear at \$2000. It will not run properly without it. Using the A command, enter the following code:

```
C000 LDA #$36
C002 STA $01
C004 LDY #$00
C006 STY $FA
C008 STY $FC
C00A STY $FE
C00C LDA #$A0
C00E STA $FB
C010 LDA #$20
C012 STA $FD
C014 LDA #$E0
C016 STA $FF
C018 LDA ($FA),Y
C01A STA ($FC),Y
C01C STA ($FE),Y
C01E INY
C01F BNE $C018
C021 INC $FB
C023 INC $FD
C025 INC $FF
C027 BNE $C018
C029 SEI
C02A LDA #$35
C02C STA $01
C02E JMP ($FFFC)
```

- 5). You may now save the code through LLMON with S "SOLAR SYSTEM",08,A000,C031.
- 6). When you load the program, you may activate it with SYS 49152. This is the decimal equivalent for \$C000, which is where we stored our transfer program.
- 7). YOU'RE DONE.

The CARTRIDGE BACKER will make a disk copy of this cartridge along with countless others. The backup procedure includes an autoboot for the disk copy so that you will not have to be concerned with the proper SYS command. Most cartridges will include some type of protection. The CARTRIDGE BACKER will remove the protection code and give you a report on the corrections. This program will save you a great deal of time.

Space prohibits an extensive discussion of the backup procedure for this cartridge. You will find cartridges of this type explained in far greater detail in the chapter on Advanced Cartridges in the PROGRAM PROTECTION MANUAL VOLUME II.

The program is WIZARD (C) 1984 by Progressive Peripherals & Software

TYPE OF PROTECTION: The disk is littered with errors. When we attempt to examine the disk with a track and sector editor, we find that accessing various tracks will cause the drive to read endlessly (ex. TRACKS 2 and 3). We also discover that track 18/0 has been modified to make the directory unlistable.

HOW TO COPY THE DISK: This program uses a non-standard format that most nibble copy programs cannot handle. You may be sure that a copier will soon be developed to handle a program of this type.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS.

PROBLEMS:

- 1). We must modify 18/0.
- 2). Code is stored under the kernal.
- 3). We must find a proper entry point.
- 4). We must write a loader program to bring our programs into memory and execute the proper SYS command.

SOLUTIONS:

Let's get to work. This program is a bit complex, but certainly not impossible. Through the unprotection of this program, we will add some useful techniques to our bag of tricks.

- 1). First, make a copy the entire disk without errors.

- 2). With your track and sector editor executed, alter 18/0 by moving the name over two bytes. Go ahead and use WIZARD for the name, but then go to HEX mode and fill in the rest of the spaces from the end of the name to the ID (WZ) with A0. This will allow us to get a proper listing of the directory.
- 3). Now that you have a proper directory, file copy all but the programs listed below to a newly formatted disk. This is the disk we will be working with.

WIZ	SPRW
LODR	M.L.
CHRW	

- 4). Load and Run the program from the original disk. When the option screen appears, insert your cartridge based machine language monitor. Reset your computer with your cartridge switch OFF. We leave the switch off so that we may preserve the code stored by the program at 8000. You should perform this reset before the program goes into the DEMO or PLAY modes. You should get the familiar blue screen showing 30719 BYTES FREE. Remove the cartridge.
- 5). Load and execute LIMON with SYS8192.
- 6). Using the I command, scroll through the memory. The first code we find is at \$1000. At 11F0, we will find the code to load the programs that we deleted from our copy disk. This is the code from the LODR program. If we use the D command, we will find numerous kernal routines. Among them, is a routine to re-format a copy disk, when it is detected by the program. Have no fear, we'll skip that part. When we return to interpret mode, we see the preparatory screens used before we enter the option screen. After a little experimentation, we find that the code at \$198C will load in a program called SCOR. This is the program that will keep track of the high scores earned in the game. When we run this section of code with a G 198C (with our copy disk in the drive), we find that we have a working version of the program. There's only one hitch, we have some valuable code stored under the KERNAL. This code controls the sprites. We'll solve that problem in a moment.
- 7). S "WIZARD",08,1000,CFFF - Do not exit the monitor yet. We will now write a short program to access the code stored under the kernal.
- 8). Through LIMON, load in HIMON. Once loaded, execute HIMON with G C000. Using the F command, fill the memory from 1000 to 9FFF with 00. This will clean things up for our next operation. We will construct our routine at 1000. Using the M command, type in the following:

```

.:1000 78 A9 35 85 01 A9 00 85
.:1008 FC 85 FE A9 E0 85 FD A9
.:1010 20 85 FF A0 00 B1 FC 91
.:1018 FE C8 C0 00 D0 F7 E6 FF
.:1020 E6 FD A5 FD D0 ED A9 37
.:1028 85 01 58 00 00 00 00 00

```

Make sure you have typed in the code correctly. Don't forget to type return after each line. When we disassemble our program at \$1000, we find the following.

1000 SEI	1017 STA (\$FE),Y
1001 LDA #\$35	1019 INY
1003 STA \$01	101A CPY #\$00
1005 LDA #\$00	101C BNE \$1015
1007 STA \$FC	101E INC \$FF
1009 STA \$FE	1020 INC \$FD
100B LDA #\$E0	1022 LDA \$FD
100D STA \$FD	1024 BNE \$1013
100F LDA #\$20	1026 LDA #\$37
1011 STA \$FF	1028 STA \$01
1013 LDY #\$00	102A CLI
1015 LDA (\$FC),Y	

This program will set the interrupt, which will keep us from locking up, and transfer the code stored under the KERNAL to \$2000. Save this program to a disk and keep it for future use. Now execute the program with G 1000. Using the I command, scroll through the code at 2000. Save this code with S "KERNAL",08,2000,4000. We may now exit the monitor.

- 9). We have a slight problem with our kernal program. When loaded, KERNAL will locate at \$2000. The program is expecting to find it at \$E000. We could write a loader program to relocate this code, or we could do it the easy way. With a track and sector editor, we can locate where the first block of the KERNAL program is stored on the disk. Go to TRACK 18 and locate KERNAL in the directory. It will be the last program in the directory and should be located on TRACK 6 SECTOR 0. Bytes 3 and 4 will tell you where the program is located. Once you have the location, go to that TRACK 6/0. Byte 3 will contain our 20. If we change this to E0, the program will relocate to \$E000 when it is loaded. The block should read as follows:

060A0020

060A - Location of the next block of the program.

0020 - Low byte - High byte of the memory address for this program. Change the 20 to an E0.

- 10). All that's left is to write a loader that will load in the two programs and SYS to 6540 (198C HEX). The following program will do the job for us:

```

10 PRINT"[CLR][3 DOWN]LOAD"CHR$(34)"WIZARD"CHR$(34)",8,1"
20 PRINT"[4 DOWN]NEW"
30 PRINT"[2 DOWN]LOAD"CHR$(34)"KERNAL"CHR$(34)",8,1"
40 PRINT"[4 DOWN]NEW"
50 PRINT"[2 DOWN]SYS6540[HOME]"
60 POKE198,5:FOR I=0 TO 4:POKE631+I,13:NEXT

```

SAVE this program to your copy disk (SAVE "WIZARD LOADER",8) and then RUN it.

11). YOU'RE DONE.

FOR THE IMPATIENT AMONG US.

When run, our unprotected program will take 30 seconds longer to load than the original. We may shorten the load time by saving the code out in pieces. The sections necessary for proper execution are as follows:

1000 - 2000
5800 - 9FFF
C000 - CFFF

With these sections and the kernal section we saved earlier, the program will run properly. Don't forget to change the loader program to accommodate the four programs.

WIZARD ADDENDUM (1) WIZARD (C) 1984 by Progressive Peripherals & Software.

After receiving a couple of calls on the Wizard procedure, we followed the steps as described in the November issue and found no problems with the procedure. This could mean that you have a different version of the program than the one we used here, or that you have left out a step in the procedure. Let's hope it's the latter. We suggest the following:

- 1). Load the program from the original disk. When the option screen appears, insert your cartridge-based machine language monitor and reset your computer, with YOUR CARTRIDGE SWITCH OFF. REMOVE THE CARTRIDGE. You should now have the blue screen. Type SYS6540 and press return with the original disk in the drive. If you get the option screen, your version of the program is probably the same as the one we used.
- 2). With that aside, now load the program you saved last month called "WIZARD". This is the main program from 1000 to CFFF. LOAD "WIZARD",8,1. Once in memory, type SYS6540 and press return. The option screen should appear. If this does not occur, you probably did not save out the main program properly. Go through steps 4-7 again and make sure that you reset with your CARTRIDGE SWITCH OFF, and that before you save out the code from 1000 to CFFF you have REMOVED YOUR CARTRIDGE MONITOR. Once the code is saved properly, repeat the test described above.
- 3). If all is well with the option screen, but you are experiencing problems with improperly formed sprites, you probably have not saved out the kernal section properly. Go through the November procedure as described, and make sure your kernal routine at 1000 is correct. Make sure you type in the 00's in 1028. Don't forget that this section of code was saved out at 2000 and must be changed on the disk to E000. Refer to the November procedure (Step 9) to alter this address.

We hope these suggestions will solve your problems. If you are still experiencing difficulties, we may only assume that you have a different version of the program.

WIZARD ADDENDUM (2) - WIZARD (C) 1984 by Progressive Peripherals & Software.

With the help of user-feedback we learned of another version of the Wizard program. We were advised by one of our users that the entry point for his version of the program was \$10FC (SYS4348) rather than \$198C. You may wish to try the procedure again with this entry point. The entire procedure can be found in the November NEWSLETTER. You may also wish to refer to the December Addendum for further clarification of the technique.

We have also learned that there seems to be a bit of confusion as to how and when to insert the cartridge-based machine language monitor. Once the protected program is running, insert your cartridge monitor with your CARTRIDGE SWITCH OFF. DO NOT insert a cartridge with the cartridge switch on. Remember, any time that you insert a cartridge into the computer with the power on you risk damage to the cartridge or computer. This is why we recommend the purchase of a cartridge board or expansion board.

The procedure for installing a cartridge switch is described in the PROGRAM PROTECTION MANUAL VOLUME I.

C.S.M. also sells a single slot cartridge board complete with a reset switch, GAME switch, EXROM switch, ENABLE switch and POWER switch.

The program is **WORDPRO 3 PLUS/64**, Copyright 1982, Professional Software Inc.

TYPE OF PROTECTION: This program will check track 1 for an error 21.

HOW TO MAKE A BACKUP COPY OF THE DISK: Copy the program with any good copy program and place an error 21 on track 1, or use a nibble copy program that will place the error for you.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

Let's do some investigating. Since this is a screen load program, it will be more convenient to alter the code on the disk. First we need to find out what is happening.

- 1). Load the program from the original disk and reset your computer after approximately 15 seconds.
- 2). Load and execute HIMON with SYS49152. Using the I command at 0800, you should easily locate the B-R for Track 1/4. You will find this in reverse order in the code. Using the D command, examine the code at 0974. There's the CMP#\$32. If we change that to CMP#\$30, our problem should be solved. But, there's a

hitch folks. If you move up to 0912 and 0915, you will find a DEC \$0974. This is where our 2 is stored for the CMP#\$32. When we change this instruction to 30, the DEC instruction for \$0974 will decrement the 30 instead of a 32. The value that is returned will lead to a program crash. We must not only change the CMP 32 to a 30, but we must eliminate the DEC instructions. Let's get to it.

- 3). Make a copy of the program disk without errors.
- 4). Load and execute your track and sector editor. We will be altering the program called "WORDPRO 3 PLUS/64. At 18/01, we find that this program is stored on the disk at 17/01. Using your file link feature, you will find the B-R at 17/14. Continuing with the program link, we find the block we will alter at 17/05. We will be looking for CE7409 (DEC\$0974). CE is the op code for decrement and 7409 is our memory address stored in low byte/high byte. We will change these six bytes to EA's (NOP). That takes care of the decrement instructions, now let's change the C932 (CMP32). Find the C932 and change the 32 to a 30. CAUTION: There is an A932 immediately following the CE instructions, do not change this instruction. We will change the C932 to C930. With these changes made, use your write feature to write the changes on the disk.
- 5). YOU'RE DONE.

The program is WORDPRO 128, Copyright 1984 by Steve Punter and Proline Software Limited.

TYPE OF PROTECTION: This program will check track 01 for an error 21.

HOW TO COPY: Most copy programs will copy this disk.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

1. Make copy of the disk without errors or file copy all the files to a formatted disk.
2. Go into 64 mode. Load and execute HIMON with SYS 49152. Clean-up the workspace with F 0800 9FFF 99.
3. L "WP128 BANK0",08. Using the M command to look at the memory area \$72E9-7309 will expose two Block-Read (B-R) commands. The program is checking track 1, sector 4 for a bad block and track 2, sector 0 for a good block. Now to alter the code.
4. We will alter four bytes. Using the M command, display and change the following bytes:

LOCATION	OLD BYTES	NEW BYTES
\$7292-93	\$F0 4C	\$EA EA
72C4-C5	D0 1A	EA EA

5. Save out the altered code to the copy disk with
S"@:WP128 BANK0",08,4000,7309.

6. That's it - you're done!

Why this program for study? Simply to illustrate that these are merely old tricks in a new package. Don't be afraid to explore a 128 program. Keep in mind that the Kernal calls are virtually the same on most Commodore (tm) machines. We made the corrections in 64 mode because that is what you are most familiar with, but we can work in 128 mode just as easily. In fact, we have a built-in monitor to assist us.

Let's examine the same program in 128 mode.

1. Enter your machine language monitor by pressing F8 (SHIFT F7).
2. Load the program, from the original disk, with L "WP128 BANK0",08.
3. Let's try that memory dump again, using M 72E9 7309. There we find the Block-Read commands again. Note that there's something different in the memory addresses on the 128 monitor. We now have a 5-digit hex number instead of the usual 4. The first digit, preceding the memory address itself, is the BANK selection (more accurately called the CONFIGURATION number to distinguish it from the RAM BANK number). We did not specify a BANK in our memory command, but we could have. Try M F 72E9 F 7309 now, which specifies bank F. As you can see, the code displayed is different. You're actually looking at part of the BASIC 7.0 ROM now.

The 128 User's Guide includes a full explanation of BANK selection. The BANK statement in BASIC 7.0 is used to select the memory configuration for the PEEK, POKE and SYS commands, just like using the 5th address digit in the monitor. BANK 0 selects RAM BANK 0 only (all 64K), while BANK 15 (\$F) brings in the KERNAL and BASIC ROMs and the I/O devices "on top" of RAM BANK 0.

4. Back to our exploration. Let's disassemble the code, using D 7246. Notice we did not specify a BANK. Bank 0 is the default in the monitor (Note: bank 15 is the default in BASIC). To get another page of the disassembly, you will have to type another D and hit RETURN. Unfortunately, you cannot scroll with this monitor. Since this is the unaltered code, we will explore the routine. The first CMP (COMPARE) instruction is located at \$7290. This is the comparison for a bad block. If the block is good we are sent to \$72E0. Continue the disassembly with D (RETURN). You will have to do this twice. At \$72C2, we have another CMP instruction. This is the check for a good block. If none is found, we are sent via the BNE to \$72E0. Step through the \$72E0 subroutine, with D 72E0. We now find a JSR to \$72CE. When we disassemble that section of code, we

find the standard KERNAL routine to output a character. At \$72E0, we find the problem area. At the end of this routine, we find a JSR \$72CE, which will send the program into an endless loop. As we said, old tricks in a new package.

5. At this point, you can make the same changes outlined in Step 4 of the 64 mode procedure to disable the protection.

Before we close the chapter on this one, a few more words on BANK selection may be in order. BANK selection allows us to choose the memory configuration we desire. This process is similar to the operation of the PLA on the Commodore 64, which is thoroughly explained in the PPM II and The Eprom Programmers Handbook. Back to the 128. On power up, the 128 defaults to BANK 15 (HEX 0F). As mentioned above, BANK 15 gives us Kernal and BASIC ROM, RAM (0), and I/O. Keep BANK selection in mind when exploring 128 programs. In this program, our job was made easier because the program title contained the BANK selection.

The program is **YOUR NET WORTH**, developed by ISA Systems Inc., Copyright 1984, Scarborough Systems Inc. This is a very good accounting type program.

TYPE OF PROTECTION: This program utilizes bad blocks as its protection scheme. There are a variety of errors on the disk designed to inhibit the copy procedure. An examination of the boot program (NW) reveals the BLOCK READ (B-R) command. The program will store the value it returns from the error channel and utilize it for the proper execution of the program.

HOW TO COPY: A copy program such as OMNICLEONE will make a working copy of this program.

HOW TO MAKE A WORKING COPY WITHOUT ERRORS ON THE DISK:

As with the previous program, we will lift a working copy from memory. We chose this method to cut down on load time. Almost 50% savings in load time may be obtained by lifting this program from memory. Examining the boot program, we find the entry point in LINE 11 (BASIC). Once the program has loaded the program modules and passed the error checking routine, it will do a SYS10900. This is the main menu screen.

1. If you have "FILL'ER UP", load and execute the program. If you do not have this program, fill memory through HIMON. Load and execute HIMON with SYS49152. Using the F command, fill memory with F 0800 9FFF 99. With this accomplished, exit to BASIC with G FCE2 (SYS64738).
2. Load the original program. Once the main menu screen appears, RESET your computer.
3. Since the program code does not extend through C Page, HIMON is still in memory. Re-enter the monitor with SYS49152.

4. Using the I command at \$0800, begin scrolling through memory to locate the end of the program code. Remember, we are looking for 99's. We find the end of the program code located at \$9C00.
5. Save out the code to a formatted disk with S "YOUR NET WORTH",08,0800,9C00. You may now file copy the HELP programs from your original disk to your formatted disk.
6. To utilize the program LOAD "YOUR NET WORTH",8,1. Once in memory, execute the program with SYS10900. Remember this was the entry point revealed to us in the boot program.
7. YOU'RE DONE!

